# An integrated index structure for object-oriented and spatio-temporal information systems

by

*Isabel M. Besembel Carrera*

Submitted in accordance with the requirements
for the degree of Doctor of Philosophy.

The University of Leeds
School of Computer Studies

September 1998

The candidate confirms that the work submitted is her own and the appropriate credit has been given where reference has been made to the work of others.

To Yohama Caribay and Ricardo Arturo, my children
To my father, always on my mind... forever in my heart
To my mother, sister and brothers....

# Acknowledgements

I am very greatful to my supervisor Dr. Stuart A. Roberts for his helpful guidance and encouragement, frienship discussions, and for carefully reading multiple drafts of this thesis.

I am also indebted to Jonás A. Montilva C. for his support, and for reading most of the chapters of this thesis. Many thanks to Nikolaus Walczuch for his interesting discussions about general indexing that I shall never forget *Niko*, and to Edgar Chacón for his help in creating a good computer environment at home to make the final corrections to this thesis. I would also like to thank all those who help me at different stages of the research, especially the members of my research group at the University of Los Andes for their patience in waiting I finish my thesis work.

Finally, I would like to acknowledge the financial support provided by the University of Los Andes.

# Abstract

An index structure is one of the access methods extensively utilized in the database area. It defines how access to the data stored in pages on disk is made by permitting the retrieval of stored objects through a defined key, that is an attribute of the object. A multi-dimensional (dD) index defines several attributes non-concatenated as its key. This work is devoted to the study of a range of multi-dimensional access methods that could solve the problem of retrieving spatio-temporal objects in object-oriented database applications. The study of some performance parameters and the comparison of five extensions to the selected index method is also included. After presenting a comparison between several dD indexing methods, the R_tree structure was selected to index spatio-temporal objects by treating homogeneously both spatial and temporal dimensions. An object-oriented development technique named TDSO was utilized to specify and design the *RTree* class extensions and the *OTree* class, which were implemented in C++.

A novel approach to building dD indexes which attempts to improve the hit ratio by accommodating spatio-temporal search operators within the indexing mechanism is the first contribution. The development of a family of methods for indexing into dD spatio-temporal data together with preliminary testing of these techniques is the second contribution of the work. We also show a substantial improvement in terms of reduction in wasted space through a redistribution policy and the achievement of better hit ratios and fewer disk accesses in some range searches. In general, we have provided support for the usefulness of the TDSO and object-oriented techniques, which led to a very flexible scheme whereby the indexing structure was easily adapted to our problem.

# Declarations

Some parts of the work presented in this thesis have been published in the following articles:

**Besembel, I. and Montilva, J. and Roberts, S.**, "A comparison between R_tree extensions in 2D", *Proceedings of the 14th. IASTED International Conference on Applied Informatics*, Hanza, M. H. (Ed.) IASTED Acta Press. (1996) 286-270.

**Besembel, I. and Roberts, S.**, "Comparison between multidimensional point and spatial access methods", *Proceedings of the World Multiconference on Systemics, Cybernetics, and Informatics. SCI'97/ISAS'97.*, Callaos, N. and Bennamoun, M. and Aguilar, J. (Eds.) (1997) 113-119.

**Besembel, I. and Roberts, S.**, "Indexing spatio-temporal objects", *Proceedings of the World Multiconference on Systemics, Cybernetics, and Informatics. SCI'97/ISAS'97.*, Callaos, N. and Bennamoun, M. and Aguilar, J. (Eds.) (1997) 120-127.

**Besembel, I. and Montilva, J.**, "Modelling spatio-temporal relationships in object-oriented applications", *Proceedings of the IASTED International Conference on Artificial Intelligence and Soft Computing. ASC'98.*, Hanza, M. H. (Ed.) IASTED Acta Press. (1998) 242-245.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

An access method defines how the access to the data stored in pages on disk is made. Indexing is one of the access methods extensively used in the database area. An index permits the retrieval of objects stored in secondary memory through a defined key which is an attribute of the object. A one-dimensional (1D) index defines a key that identifies the object completely, called a primary key. A multi-dimensional (dD) index defines several attributes non-concatenated as its key. Each attribute is called a secondary key. Relational databases typically use the B_tree or a variant called the B+_tree [BM72], [BU77], [Com79], [Bes84] data structure to index and cluster data within the tables supporting a particular database application. The B_tree structure is the more commonly used 1D index giving the best performances in this category of indexes. This structure has also been utilised as a dD index when several index attribute values are concatenated to have one only value as the index key for B_tree indexing.

The B_tree index has many very good characteristics, that are very desirable in a multi-dimensional index [Fre93], such as :

1. the tree is always balanced;
2. worst-case single-object search, insertion (excluding overflow) and deletion (excluding underflow) require no more disk access than the height of the tree. Thus, the access time for a single object is constant and predictable, for a given tree size;
3. overflow (split) and underflow (merge) propagate only upwards in the tree;

4. high average leaf and branch node occupancies, more than 67%;
5. worst-case node occupancy near 50%;
6. the index size is always directly proportional to the quantity of indexed objects;
7. the method is fully dynamic, no reorganizations are required.

Object-oriented databases have been developed using B_trees to index their objects with a number of different strategies [BK89], [MS90], i.e. the hierarchical index which indexes objects by using their object identifier concatenated with their class number in the system.

We are interested in the study of a diversity of dD access methods that could fulfill B_tree's main characteristics and could solve the problem of retrieving spatio-temporal objects in object-oriented database applications.

## 1.1 Motivation and research objectives

Originally, we were looking for a good index method to index multimedia, object-oriented, and spatial objects to be used in the D/K model proposed by a colleague in his PhD thesis [Mon93]. By studying the D/K model we realise that it specifies spatial classes such as topological classes (SpChain, SpNode, and SpPolygon), spatial object classes (SpatialObject, SpatialRepresentation, SpWindow, and Map), spatial graph classes (directed and undirected), etc. Figures 1.1 and 1.2 show a summary of this model, where the SpatialObject class has one attribute called *representations* that refer to another class named SpatialRepresentation class. This means that any spatial object in the system has to be reachable from the SpatialObject class and has to have several representations based on scales, geometry, and position attributes. This model was proposed to manage two spatial dimensions and it also supports graphical objects. A graphical object has attributes as coordinates (a sequence of points), a bounding box (mbr), and drawingDot (a drawing pattern). The relationship between spatial and graphical objects is made by the geometry defined for the SpatialObject class. This geometry is specified by a set of objects of the GraphicalObject class. Thus, any spatial object has a point representation as well as an mbr representation in the D/K model. In our approach, we take these two representations as a base to develop our work where the spatial object shape is approximated by a minimum bounding box, and the index structure does not exactly evaluate a query, but only yields a set of candidates, that may fulfill the query condition.

**Basic classes**

| **Magnitude classes** | **Collection** | **Graphical** |
|---|---|---|
| Character | ArrayOf[X] | Form (Image) |
| Date | String / Text | Point |
| Time | SetOf[X] | Rectangle |
| Integer | ListOf[X] | Paragraph |
| Float | Dictionary[X,Y] | |
| Association | OrderedCollectionOf[X] | |

**D/K multimedia classes**

| **Textual** | **Graphical** | **Image** | **Hypermedia** |
|---|---|---|---|
| Text | GraphicalPoint | Image | Unit |
| Paragraph | Line | | Link |
| | Polygon | | Button |
| | Window | | HyperNetwork |

Figure 1.1: Basic and multimedia classes of the D/K model.

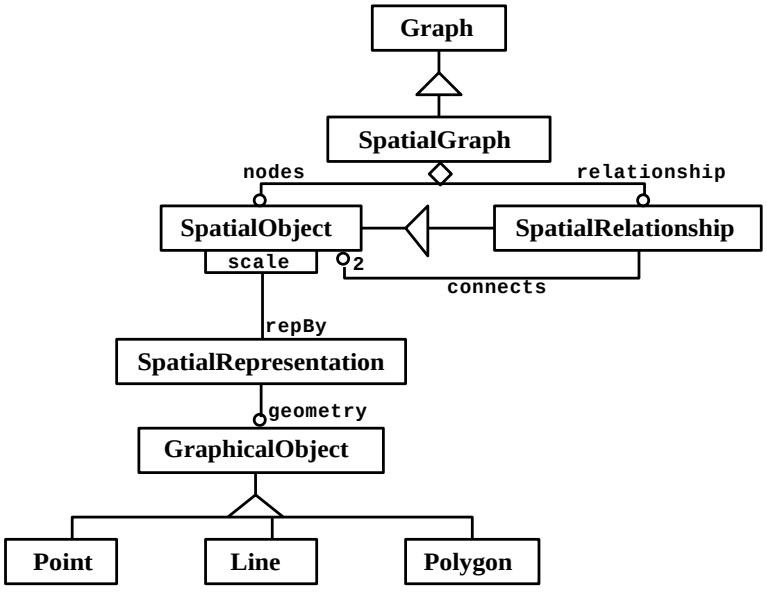



Figure 1.2: D/K classes.

Researchers working on graphic and on geographic applications have created many dD index structures to be used in their research areas. One of them is the quadtree [SW88], [Sam89], [FB74], that was specially created to manage raster data in two dimensions (2D) with good performance. However, the D/K model supports vector instead of raster representation and we focus our attention in spatial indexing for vector data.

At the same time, researchers in the database area, working on non-standard database applications as Geographic Information Systems (GIS), Computer Aided Design (CAD) or Computer Aided Manufacturing (CAM), among others, have proposed other index structures, such as : the R_tree [Gut84], the $R^+$_tree [SRF87], the cell tree [Gun89], [GB91], the grid file [NHS84], etc. These data structures were cataloged as dD access methods and they work mainly on vector data. They are named multi-dimensional because they use several non concatenated index keys to index and cluster data. These several index keys are data attributes pertaining to real space or not depending on the application. In the case of pertaining to real space, i. e. geographic coordinates, the index structure is normally called a *spatial* index [SW93], otherwise it is named a *multi-criteria* index.

Multi-dimensional access methods can be divided into multi-dimensional point access methods (MPAM), and spatial access methods (SAM). MPAM works with dD points that are indexed by regions of points where each region is stored in the same data page. Example index structures representing MPAM are K-D-B tree, grid file, and buddy tree. On the other hand, SAM organises spatial objects according to spatial position and extension, normally indexed by the minimal bounding rectangle (mbr). The spatial dimensions are basically space coordinates, 2D or 3D. Example SAM structures are the spatial k-d tree, the R_tree, $R^+$_tree, and cell tree.

Non standard database applications may include a temporal domain together with a spatial one. Spatio-temporal databases can be used for these kind of applications, and each time more people realise the advantages of having temporal data in the database. Actually, CAD/CAM software includes a kind of temporal management in its version manager. To deal with this combination of spatial and temporal data, it is desirable to have specific access methods to obtain better system performance. These access methods are called spatio-temporal, and two of them are the RT_tree [XHL90] and the TR*_tree [SK92]. A spatio-temporal object may change its spatial location and/or shape at different time intervals. Therefore, the object remains the same but its spatio-temporal state changes.

The two methods mentioned above are both based on the R_tree index. It is

interesting because the R_tree is a generalization of the B_tree to higher dimensions. Therefore, the R_tree is a dD access method which has the majority of the good characteristics already defined for the B_tree. The only one that R_tree does not maintain is that mentioned in point 2, where it requires more disk accesses than the height of the tree for worst-case single-object search, insertion (without overflow), and deletion (without underflow). This last characteristic is due to the presence of overlapping regions in the non leaf nodes of the R_tree.

The approach of the work by X. Xu et al. in [XHL90] is to include temporal attributes in the index key along with the spatial attributes. This is the approach adopted in the thesis.

Databases and indexing are two areas narrowly related because the first one mainly uses index structures to manage objects stored on disk. Object-oriented databases (OODB) integrate concepts from the object-oriented approach with database capabilities. To manage objects in OODB, an identifier is assigned to each object in the database, named ObjId which is used as the indexing key. There are several kind of indexes in this area, but all of them are based on the B_tree. The ObjId is the primary key and requires a 1D index structure, although it is possible to use a dD index if keys other than the ObjId are included.

Spatial databases contain spatial objects and define spatial object location as its index key, but this key is composed of many values normally representing X and Y coordinate values. Therefore by nature, the spatial key is multi-dimensional and all of the spatial indexes that use this index key are multi-dimensional too.

In the case of temporal databases, temporal objects have at least one temporal attribute. If several time dimensions are considered like valid time and transaction time, then a dD index is required to manage these dimensions separately. But if only one time dimension, then the B_tree is the best 1D index method.

We are interested in the study of a diversity of dD access methods to decide on a method which permits spatial, temporal, and spatio-temporal retrieval and operations of spatial, temporal or spatio-temporal objects managed by following the object-oriented approach as our first objective in this work.

After choosing the R_tree structure, we were interested to explore and test possible index class extensions to try to obtain better performances in retrieving spatial, temporal, and spatio-temporal objects as our second objective.

## 1.2 Problems and proposed solutions

In this section, we describe some problems related to spatio-temporal indexing. First of all, we focus on the spatial object retrieval by an index method which attempts to avoid the extra geometric calculus due to object shape. Secondly, we study the spatio-temporal indexing problem by using a dD structure and treating both metrics homogeneously. Finally, we deal with the problem of testing some of the solutions we propose in this work.

### 1.2.1 Spatial indexing without extra geometric calculus

The spatial access methods present the general problem of retrieving spatial objects yet avoiding the extra geometric calculus due to object shape. This problem is impossible to solve without object partitioning which we wish to avoid, because object partitioning divides spatial objects into several pieces that are indexed separately. For retrieving one or several spatial objects, more index searches must be performed than those needed without object partitioning, with a corresponding larger search time. Additionally, there is not a fixed number of pieces for partitioning a specific spatial object making it impossible to calculate an average search time for spatial objects.

We propose a new index structure named the O_tree, which defines two containers to index spatial objects. One of them is the mbr and the other one is another mbr built rotated 45 degrees in relation to the X-Y axes. These two containers form an *irregular* octagon that better defines the object shape in case of polygons, and permits less extra geometric calculus in spatial queries. But, the original problem persists because object shapes can take any a variety of forms. A good approach to try to solve this problem is the TR*_tree where objects are partitioned in several trapezoids and each of them are indexed separately. We do not follow up this solution because of object partitioning.

### 1.2.2 Spatio-temporal indexing

We define a spatio-temporal object as an object that has at least one spatial and one temporal property. It can be represented by a four tuple containing *(ObjId, spatialAtt, temporalAtt, att)*, where *ObjId* is the object identifier, *spatialAtt* is the spatial location and/or shape, *temporalAtt* is the temporal property, and *att* represents other properties different from those spatial and temporal.

The problem of indexing spatio-temporal objects is then a problem of defining a dD index, because both spatial and temporal metrics of the object are multi-dimensional. We propose the object-oriented R_tree extended with spatial, temporal, and spatio-temporal search operators. In this extension, we consider three groups of relationships among objects, namely: spatio-temporal, spatial, and temporal. In spatio-temporal, we only consider the topological relationships among $d$ dimensional rectangles, where spatial and temporal dimensions are treated homogeneously. The $d$ dimensions defined for these objects are internally represented by using a unique format unifying data representation for homogeneous treatment. We also consider multiple combinations of the defined search operators finding a minimal set of them. A brief summary of this work is presented in [BR97b].

To implement the R_tree index, we choose the object-oriented approach building the RTree class in C++. This class was formally specified and designed obtaining five extensions, where each extension differs from the other in some aspect to the implementation. These extensions were made to test which one performs best after a comparison of its results. Because of the facilities of the object-oriented approach, we built the O_tree, which is our proposed index method based also on the R_tree. We call it the sixth extension, where we only modify some functions of the R_tree original class, and delete all temporal and spatio- temporal functions only conserving the spatial ones.

In order to solve the problem of choosing the best extension that we have, experiments based on four different kind of data were designed. Real data was obtained from some institutions, but we did not find appropriate data to run and test some of the extensions. The real data available was preprocessed obtaining a compacted data set, but was insufficiently demanding in terms of the quantity required to test the extension fully. Therefore, we generated datasets for points uniformly distributed in 2D and 3D space and in space-time, respectively. Several important parameters were defined, in order to compare the extensions and make conclusions about which is the best extension we had.

### 1.2.3 D/K spatio-temporal objects indexing

We extend the original D/K model to support two more classes of objects, temporal and spatio-temporal, respectively. The original proposition about an integrated index structure [Bes93], which supports multimedia, spatial, and object-oriented information systems was changed to one supporting spatio-temporal objects too.

Thus, each spatial object has several representations based on scales, geometry, and position attributes. This latter attribute is represented by a point. It suggests the need of a MPAM for indexing spatial objects in the D/K model. Nevertheless, this model uses graphical objects too containing an mbr attribute used to represent the geometry of the associated spatial object. As we say before, any spatial object has a point representation as well as an mbr representation in the D/K model. In conclusion, the choice of an indexing method was not trivial, but it had to be a method that can support both point and spatial indexing. We found in the R_tree a solution because its entries are written as coordinate intervals where a coordinate point can be represented by an interval of zero length.

As we mentioned before, the D/K model follows the object-oriented approach, therefore an index method is needed to treat object identifiers and another index to support spatio-temporal objects. It suggests the use of an access method that separates the spatio-temporal index itself from the data object file. Thus, the cell tree cannot be used without modifying its conception, and the R_tree may be used due to its good characteristics and its adequacy to this problem.

## 1.3 Organisation of the thesis

The rest of this work is presented as follows: Chapter 2 describes the principal multi-dimensional access methods, highlighting the importance of object indexing and discussing the main characteristics of each of the presented index structures. It begins with some multi-dimensional point access methods, followed by those methods called spatial access methods, and finishing in a summary of both types of multi-dimensional indexing methods, shown in a comparative table. It also includes two spatio-temporal indexing methods, both based on the R_tree structure. A brief summary of these structures is included in [BR97a].

The main spatio-temporal concepts used in spatio-temporal object-oriented databases is specified in chapter 3. These concepts are presented beginning with the more general object-oriented concepts related to objects, relationships, and object-oriented databases. After that, is described spatial objects, and temporal ones. Finishing with spatio-temporal objects, relationships, and indexing characteristics. We also mention the approach to extending of the D/K model to support spatio-temporal objects that will be appear in [BM98].

Chapter 4 presents an object-oriented formal model of the R_tree structure. At the beginning, we include some design characteristics to better explain the bases of

our formal model, and we finalize with the presentations of the requirements for the **RTree** class extensions.

An object-oriented R_tree design and implementation is explained in chapter 5. It contains the specification and implementation of the **RTree** class by using the Object System Development Technique (TDSO) [Bes95]. It also specifies R_tree extensions based on the original one which is extension 1, and we discuss in detail the several possible spatio-temporal operators to support spatio-temporal relationships. A paper where is presented the first four extensions is reported in [BMR96].

Chapter 6 contains a description of the kind of data that we use to test the R_tree extensions, the experimental design, and a discussion of the results obtained.

The conclusions and future work of this thesis are described in chapter 7. A brief summary of the main characteristics of some Object-Oriented Database Management Systems is presented in appendix A, including a comparative table. Appendix B contains the rest of the TDSO implementations corresponding to each extension of the tree. Finally, some results obtained from the experiments, and a description of the kind of files used to test extension 6, are included in appendix C.

# Chapter 2

# Multi-dimensional access methods

Access methods are data structures that determine how the access to the data stored in pages on disk is made. One of these access methods is named indexing. In this thesis we are concerned with data access based on multidimensional search criteria, such as multidimensional point access methods (MPAM) and spatial access methods (SAM). A MPAM partitions the data space into regions, such that all records in one region are stored in the same data page. A SAM organises spatial objects according to spatial position indexed by their minimal bounding rectangles (*mbr*). The main structures that represent MPAM are K-D-B tree, grid file, and buddy tree; those representing SAM are spatial k-d tree, R_tree, $R^+$_tree, and cell tree. We summarize the main characteristics of each access method and we present a table comparing the main structures in terms of several properties such as nodes of the structures, order, spatial technique, spatial object, insertion and deletion of data objects, handling of objects, fragmentation.

An index permits the retrieval of objects stored in secondary memory through a defined key which is an attribute of the object. This key may be either one attribute that identifies the object completely (called the primary key), or several attributes that, concatenated, form a primary key, or several attributes non-concatenated (taken separately). In the last case, each attribute is called a secondary key of the object. In this chapter, we are interested in those access methods that permit object retrieval by secondary keys. These methods are known as multikey, multiattribute, or multidimensional access methods as well.

Our main objective is to study a variety of these methods to decide on a method which permits spatial retrieval and operations on spatial and/or temporal objects. Following the object-oriented approach, a spatial object must belong to a class of spatial objects which has as its instances the whole of the objects that contain spatial properties. By spatial properties we refer to geometrical (shape, position, and size) and topological (adjacency, connectivity, and inclusion) properties, where we shall assume the topological properties can be derived from the geometrical ones[1]. The main retrieval operation of spatial objects is made by location in space, and the majority of the spatial access methods take the position or some property related to position, as the defined key. Some methods developed for indexing raster data (e.g. quadtrees) may also be applied to vector representation, but do not consider these here. We are also interested in temporal access methods for the mixed model, and more specifically in methods that could treat objects varying in space and time. We define a temporal object class which contains all time varying objects with temporal attributes. These objects are mainly retrieved by the temporal attributes which are used to index them. Temporal is a generic term implying some type of time support in the database.

To build and present each access method, we use some real data shown in figure 2.1. The first column named *ObjId* is the object identifier, the second one is the minimal bounding rectangle (*mbr*) which actually covers that object, the third one represents other data relevant to the application, and the last column indicates the object's centroid. This last column is needed to build only one of the indexing methods mentioned below. Objects covered by *mbr* are parcels of land at Carora city.

Section 2.1 below gives a description of some multidimensional point access methods basically summarized in terms of their main characteristics which are presented in the cited bibliography of each method. Section 2.2 presents, in a similar manner, concepts and data structures used in spatial access methods. Section 2.3 summarizes in a comparative table the access methods named in the two previous sections including a brief description of each property that is mentioned in the table. Two main spatio-temporal access methods are presented in section 2.4. Finally, some comments and conclusions are included here as well as a tabular comparative analysis.

[1]This ignores the issue that topological relationships based on geometrical may change with geometric scale.

| ObjId | *mbr* | Other data | Centroid |
|---|---|---|---|
| O1 | 318,124:374,195 | 66 | 346,159.5 |
| O2 | 287,124:318,147 | 73 | 302.5,135.5 |
| O3 | 326,117:382,188 | 66 | 354,152.5 |
| O4 | 252,151:282,175 | 73 | 267,163 |
| O5 | 287,147:345,220 | 66 | 316,183.5 |
| O6 | 305,147:454,215 | 50 | 379.5,181 |
| O7 | 239,139:272,168 | 73 | 255.5,153.5 |
| O8 | 481,127:535,194 | 50 | 508,160.5 |
| O9 | 454,127:481,147 | 60 | 467.5,137 |
| O10 | 522,327:540,341 | 50 | 531,334 |
| O11 | 423,172:476,241 | 55 | 449.5,206.5 |
| O12 | 382,166:410,188 | 60 | 396,177 |
| O13 | 449,152:503,219 | 50 | 476,185.5 |
| O14 | 423,152:449,172 | 60 | 436,162 |
| O15 | 282,151:340,225 | 66 | 311,188 |
| O16 | 391,198:444,266 | 55 | 417.5,232 |
| O17 | 345,195:374,220 | 60 | 359.5,207.5 |
| O18 | 507,194:535,215 | 50 | 521,204.5 |
| O19 | 418,176:473,244 | 55 | 445.5,210 |
| O20 | 391,176:418,198 | 60 | 404.5,187 |

Figure 2.1: Some real data of Carora city of Venezuela.

## 2.1 Multidimensional point access methods

The main characteristic of a multidimensional point access method (MPAM) is that it partitions the data space into regions, such that all records in one region are stored in the same data page. The division of the whole space into disjoint subregions is normally made depending on the number of points in each subregion. As pointed out by M. W. Freeston in [Fre89]:

"(Grid files)"

> '...adopt a direct geometric representation of the data: the tuples of an n-ary relation are represented as points in an n-dimensional hyperspace, the dimensions of which are the domains of the n attributes. The problem is how to allocate the tuples in this hyperspace to the set of disk blocks (or pages) in the file. In geometric terms the obvious thing to do is to divide up the data space into a set of hyper-rectangles or block regions, each of which corresponds to a disk page. However, in order to maximize the storage efficiency of the file, the number of such regions, and the positions of their boundaries, have to be arranged so that each of the corresponding disk pages has a high data occupancy.'

There are three important attributes for classifying MPAM according to T. Sellis et al. in [SRF87], they are i) whether the position of the splitting hyperplane is

fixed or not, ii) the splitting dimensionality, which can be only 1-dimension or all d-dimensions as the quad/oct-trees, and iii) the locality of the hyperplane, which can affect all the regions in the given direction or solely inside the region to be split.

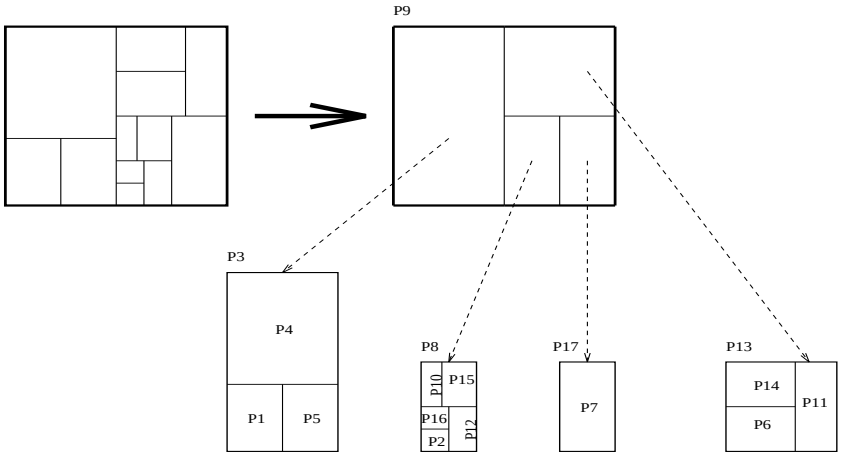
### 2.1.1 K-D-B tree

This structure was presented by J. T. Robinson in [Rob81] as a large, dynamic, multikey index based on a combination of the k-d trees [Ben75], and the B_trees. The k-d tree is a multidimensional binary tree, where each level corresponds to a dimension that is chosen cyclically among of the defined dimensions of the binary tree.

It does not mention any spatial technique, but it defines a *point* as a value $x_i$ in a set formed by the cartesian product of a list of domains, and a *region* as a set of all points $(x_0, x_1, \ldots, x_{K-1})$ satisfying $min_i \leq x_i < max_i$, $0 \leq i \leq K-1$. The concept of *mbr* is not mentioned, but it is used in the structure by means of region pages. Thus, region pages contain a collection of entries of the form: (region, pageId), and point pages have entries of the form: (point, location), where pageId is a page identifier, and location is a record identifier.

There are no overlapping regions because regions in the page are disjoint, and their union is a region. The choice of the splitting dimension in a region page depends on the cycle of partitioning which is a concept taken from k-d trees [Ben75]. In underflow conditions (i.e. when a page has less than the half of the maximum number of entries), reorganization is not used in Robinson's paper [Rob81], although an outline algorithm for this purpose is given. There is no performance comparison with other methods because the K-D-B tree is the first multidimensional structure after the k-d trees of Bentley.

An example of this structure is presented in figure 2.2. It was built by following the algorithms presented by J. T. Robinson, with a dimension of 2 (K=2), a region page capacity (c) of 5, and a point page capacity (b) of 4. Figure 2.2 also shows both region pages and point pages formats. Each page is identified by a number followed by the height of this page, the current number of entries, a dimension identifier for the last split in this page, and the corresponding entries. Symbols **L** and **H** mean lowest and highest numbers in each dimension, respectively. When the page size is fixed, region pages have less capacity than point pages, because entries in region pages are longer than entries in point pages. This example shows a percentage of occupancy per page of 76.71%




Region pages of the tree

| Page Id. | Node height | Number of entries | Dimension of the last split | Region node entries | |
|---|---|---|---|---|---|
| P3 | 1 | 3 | Y | L,318:L,151:P1 | 318,326:L,151:P5 |
| | | | | L,326:151,H:P4 | |
| P8 | 1 | 5 | Y | 326,423:L,176:P2 | 326,423:176,188:P16 |
| | | | | 423,481:L,188:P12 | 326,391:188,215:P10 |
| | | | | 391,481:188,215:P15 | |
| P9 | 2 | 4 | X | L,326:L,H:P3 | 326,481:L,215:P8 |
| | | | | 481,H:L,215:P17 | 326,H:215,H:P13 |
| P13 | 1 | 3 | X | 326,522:215,225:P6 | 326,522:225,H:P14 |
| | | | | 522,H:215,H:P11 | |
| P17 | 1 | 1 | Y | 481,H:L,215:P7 | |

Point pages of the tree

| Page Id. | Node height | Number of entries | Dimension of the last split | Point node entries | | |
|---|---|---|---|---|---|---|
| P1 | 0 | 4 | Y | 287,124:O2 | 239,139:O7 | 287,147:O5 |
| | | | | 305,147:O6 | | |
| P2 | 0 | 3 | X | 326,117:O3 | 382,166:O12 | 449,152:O13 |
| P4 | 0 | 4 | X | 252,151:O4 | 272,168:O7 | 282,151:O15 |
| | | | | 282,175:O4 | | |
| P5 | 0 | 2 | Y | 318,124:O1 | 318,147:O2 | |
| P6 | 0 | 4 | X | 454,215:O6 | 503,219:O13 | 345,220:O5 |
| | | | | 374,220:O17 | | |
| P7 | 0 | 4 | Y | 481,127:O8 | 481,147:O9 | 507,194:O18 |
| | | | | 535,194:O8 | | |
| P10 | 0 | 3 | Y | 345,195:O17 | 374,195:O1 | 382,188:O3 |
| P11 | 0 | 3 | Y | 522,327:O10 | 540,341:O10 | 535,215:O18 |
| P12 | 0 | 4 | Y | 454,127:O9 | 423,152:O14 | 423,172:O11 |
| | | | | 449,172:O14 | | |
| P14 | 0 | 4 | X | 340,225:O15 | 476,241:O11 | 444,266:O16 |
| | | | | 473,244:O19 | | |
| P15 | 0 | 3 | X | 410,188:O12 | 391,198:O16 | 418,198:O20 |
| P16 | 0 | 2 | X | 391,176:O20 | 418,176:O19 | |

Figure 2.2: A 2-D-B tree with 5 entries/region page, and 4 entries/point page.

### 2.1.2 Grid file

The grid file was first presented by J. Nievergelt et al. in [NHS84] as a multidimensional, symmetric, and adaptable file structure for handling large amounts of multidimensional data. It partitions a k-dimensional data space into k unidimensional arrays called scales, each representing a single dimension. Scales are indices of a k-dimensional dynamic array named the grid directory, and each element of this directory is a pointer to a disk block called a bucket. Several directory cells can share a bucket, and the region defined in this way is called a bucket region, which has a rectangular shape. It supports the two-disk-access principle for single point retrieval. Thus, an exact match search is made by searching in each scale for the correct subindex for each dimension, without any disk accesses, after that, subindices determine the correct directory cell where the bucket address is located. A second access reads from disk the bucket containing the required information.

An extension of this structure presented by D. Hernández in [Her90] shows a grid index that follows the same principles for the scales and directory, but the bucket format is different. In this section, we use the grid index to show an example of a grid file built from the data of figure 2.1, where bucket entries comprise a deletion flag for that entry, an entry point, and the object reference. A bucket is composed of a bucket identifier and a set of bucket entries. Buckets contain references to objects which are stored on other file. This extension cannot support the two-disk-accesses principle, but it can be used as a multiattribute index method. Figure 2.3 shows the example mentioned above. A grid file was built in a similar manner to the previous structure, but each point in the *mbr* of an object is indexed separately because the grid index is not spatial. A highest value (HV) is used for each scale to make algorithm shorter. The percentage of bucket occupancy is 76.92%.

### 2.1.3 Buddy tree

This structure was presented by B. Seeger and H. -P. Kriegel in [SK90] as a new multidimensional access method supporting point as well as spatial data. It is a combination of the R_tree and the grid file. Its main characteristic being that it avoids the partitioning of empty data space. Its name was taken from its partitioning and reorganization strategies based on a generalization of the buddy system, which is obtained when a page is partitioned in two forming two new subregions of the original one. In a buddy system, each page and its region have a unique buddy from which it is split off.

Scale Y

| <151 | <188 | <195 | <215 | <HV |
|---|---|---|---|---|

Scale X

| | | | | | |
|---|---|---|---|---|---|
| <318 | P1 | P3 | P3 | P3 | P3 |
| <326 | P4 | P3 | P3 | P3 | P3 |
| <423 | P5 | P2 | P9 | P12 | P6 |
| <449 | P5 | P13 | P9 | P12 | P6 |
| <522 | P5 | P10 | P9 | P12 | P11 |
| <HV | P5 | P8 | P8 | P8 | P7 |

Directory

| Page Id. | Bucket entries<br>Deletion flag, point, and object reference | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| P1 | 0 | 287,124:O2 | 0 | 287,147:O5 | 0 | 305,147:O6 | 0 | 239,139:O7 |
| P2 | 0 | 382,166:O12 | 0 | 391,176:O20 | 0 | 418,176:O19 | | |
| P3 | 0 | 252,151:O4 | 0 | 282,175:O4 | 0 | 272,168:O7 | 0 | 282,151:O15 |
| P4 | 0 | 318,124:O1 | 0 | 318,147:O2 | | | | |
| P5 | 0 | 326,117:O3 | 0 | 481,127:O8 | 0 | 454,127:O9 | 0 | 481,147:O9 |
| P6 | 0 | 340,225:O15 | 0 | 345,220:O5 | 0 | 444,266:O16 | 0 | 374,220:O17 |
| P7 | 0 | 522,327:O10 | 0 | 540,341:O10 | 0 | 535,215:O18 | 0 | |
| P8 | 0 | 535,194:O8 | | | | | | |
| P9 | 0 | 382,188:O3 | 0 | 410,188:O12 | 0 | 507,194:O18 | 0 | |
| P10 | 0 | 449,152:O13 | 0 | 449,172:O14 | | | | |
| P11 | 0 | 454,215:O6 | 0 | 476,241:O11 | 0 | 503,219:O13 | 0 | 473,244:O19 |
| P12 | 0 | 345,195:O17 | 0 | 374,195:O1 | 0 | 391,198:O16 | 0 | 418,198:O20 |
| P13 | 0 | 423,152:O14 | 0 | 423,172:O11 | | | | |

Figure 2.3: A grid index with 4 entries/bucket.

Insertions and deletions are restricted to exactly one directory path. The directory grows linearly with the number of file records. The data space is partitioned into *mbr*s of the actual data, and the rectangles in directory pages are disjoint. It does not allow overflow pages and it is not dependent on the sequence of data insertions.

The nodes consist of a collection of entries where each entry is a tuple of $R_i$, which is a k-dimensional rectangle and $p_i$ which is a pointer referring to a subtree or to a data page. The rectangles in the directory nodes must be a regular B-partition of the data space which is defined as follows: A B-rectangle of a k-dimensional rectangle **S** is another k-dimensional rectangle **R** which is generated by successive halving of **S**. For an arbitrary rectangle $R \subseteq D$, where **D** is the data space, there exists a smallest B-rectangle of **D** such that $R \subseteq B$. The B-rectangle of **R** is called the B-region of **R**, in short **B(R)**. Thus, a set of k-dimensional rectangles $\{R_1, \ldots, R_k\}, k \geq 1$, is called a B-partition of the data space **D** if and only if $B(R_i) \cap B(R_j) = \emptyset \forall i, j \in \{1, \ldots, k\}, i \neq j$.

For merging two pages in the buddy tree, the regions of the pages must be buddies, which is formalized as follows,

> Let $V = \{R_1, \ldots, R_k\}$ be a B-partition, $k > 1$, and let $S, T \in V, S \neq T$. The rectangles $S, T$ are called buddies if and only if $B(S \cup T) \cap B(R) = \emptyset$, $\forall R \in V \setminus \{S, T\}$.

That is, two rectangles are buddies provided the B-region enclosing them does not intersect with the B-region of any of two other rectangles.

For splitting a directory page, the B-partition of the buddy tree has to be a regular B-partition. Thus, a B-partition $V = \{R_1, \ldots, R_k\}, k \geq 2$ is called regular, if and only if all B-rectangles $B(R_i), 1 \leq i \leq k$ can be represented as a kd-trie. A kd-trie is a binary digital tree where the internal nodes contain an axis and two subtree pointers, and the leaves have the rectangles of a B-partition.

The directory of the buddy tree is unbalanced and this property is the reason why it guarantees a linear growth of the directory in the number of file records. A buddy tree built with two dimensions, 5 entries/directory pages, and 4 entries/data pages is shown in figure 2.4. Directory pages are also shown in the figure, where each page has a page identifier, the actual number of entries in a page, and the node height or page level. A directory page entry is composed of an *mbr* and a page reference. Data pages are leaf nodes of the buddy tree. Figure 2.4 was obtained through following the directions presented in the original paper describing the buddy

tree [SK90]. Surprisingly, each time we have to split a node, the paper indicated that the Y dimension should be used for splitting, so we probably obtained a special case that is not very representative of the buddy tree shape for space partitioning. The percentage of page occupancy is 69.41%.

## 2.2 Spatial access methods

Spatial access methods (SAM) are access methods for organizing multidimensional spatial objects, like rectangles, polygons, etc. They are also known as methods for rectangles too. According to T. Sellis et al. in [SRF87], we can classify these methods into three categories such as

> Methods that transform rectangles into points within a space of higher dimensionality. Examples: the k-d trees, and the grid file.
> Methods that use space filling curves, to map a k-d space onto a 1-d space. The idea is to transform k-dimensional objects to line segments using the z-transformation.
> Methods that divide the original space into appropriate subregions (overlapping or disjoint). Examples: the R_tree, the $R^+$_tree, and the cell tree.

### 2.2.1 Spatial k-d tree

The spatial k-d tree (sk-d tree) proposed by B. C. Ooi et al. in [OSDM91] is a structure based on k-dimensional binary trees (k-d tree) presented by J. L. Bentley in [Ben75]. According to its authors its main characteristics are that it uses object bounding as a spatial technique that avoids both object duplication and object division. Thus, it uses *mbr*s for indexing spatial objects. An *mbr* is defined as an array of a single dimension containing the centroid, and extensions in each of the k directions. It supports both intersection and containment search.

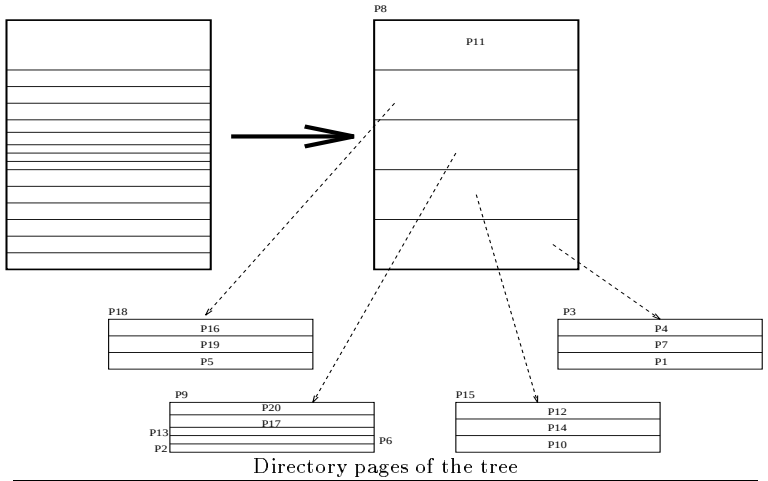
At each node, a discriminator value is chosen in one of the dimensions to partition a k-dimensional space into two subspaces, the high (HISON) and the low (LOSON) subspaces. It has two types of nodes, the internal nodes are of the form:

$$(discriminator, max_{LOSON}, disc-value, loson-ptr, min_{HISON}, hison-ptr),$$

and leaf nodes are of the form:

$$(bound, min-range, max-range, page-ptr),$$

where




Directory pages of the tree

| Page Id. | Number of entries | Node height | Directory node entries | |
|---|---|---|---|---|
| P3 | 3 | 1 | 287,117:326,117:P1 | 287,124:481,127:P7 |
| | | | 239,139:239,139:P4 | |
| P8 | 5 | 2 | 239,117:481,139:P3 | 252,147:481,166:P15 |
| | | | 279,168:535,198:P9 | 340,215:535,244:P18 |
| | | | 444,266:540,341:P11 | |
| P9 | 5 | 1 | 272,168:449,172:P6 | 282,175:418,176:P2 |
| | | | 382,188:410,188:P13 | 345,194:535,195:P17 |
| | | | 391,198:418,198:P20 | |
| P15 | 3 | 1 | 305,147:481,147:P10 | 252,151:282,151:P14 |
| | | | 382,152:449,166:P12 | |
| P18 | 3 | 1 | 454,215:535,219:P5 | 340,220:374,225:P19 |
| | | | 473,241:476,244:P16 | |

Data pages of the tree

| Page Id. | Number of entries | Node height | Data node entries | | | |
|---|---|---|---|---|---|---|
| P1 | 2 | 0 | 287,117:O5 | 326,117:O3 | | |
| P2 | 3 | 0 | 282,175:O4 | 418,176:O19 | 391,176:O20 | |
| P4 | 1 | 0 | 239,139:O7 | | | |
| P5 | 3 | 0 | 454,215:O6 | 503,219:O13 | 535,215:O18 | |
| P6 | 3 | 0 | 272,168:O7 | 423,172:O11 | 449,172:O14 | |
| P7 | 4 | 0 | 287,124:O2 | 318,124:O1 | 481,127:O8 | 454,127:O9 |
| P10 | 3 | 0 | 305,147:O6 | 318,147:O2 | 481,147:O9 | |
| P11 | 3 | 0 | 522,327:O10 | 540,341:O10 | 444,266:O16 | |
| P12 | 3 | 0 | 382,166:O12 | 449,152:O13 | 423,152:O14 | |
| P13 | 2 | 0 | 382,188:O3 | 410,188:O12 | | |
| P14 | 2 | 0 | 252,151:O4 | 282,151:O15 | | |
| P16 | 2 | 0 | 476,241:O11 | 473,244:O19 | | |
| P17 | 4 | 0 | 535,194:O8 | 507,194:O18 | 374,195:O1 | 345,195:O17 |
| P19 | 3 | 0 | 345,220:O5 | 374,220:O17 | 340,225:O15 | |
| P20 | 2 | 0 | 391,198:O16 | 418,198:O20 | | |

Figure 2.4: A buddy tree with 5 entries/directory page, and 4 entries/data page.

*discriminator* indicates the dimension that is being partitioned.

$disc - value$ is the value that partitions the space.

$max_{LOSON}$ is the maximum range value of the LOSON subspace.

$min_{HISON}$ is the minimum range value of the HISON subspace along the dimension specified by *discriminator*.

*bound* is the dimension.

$min - range$ is the minimum value of objects in the data page.

$max - range$ is the maximum values of objects in the data page.

$page - ptr$ is the address of a page in which *mbr* and identifiers of the object are stored.

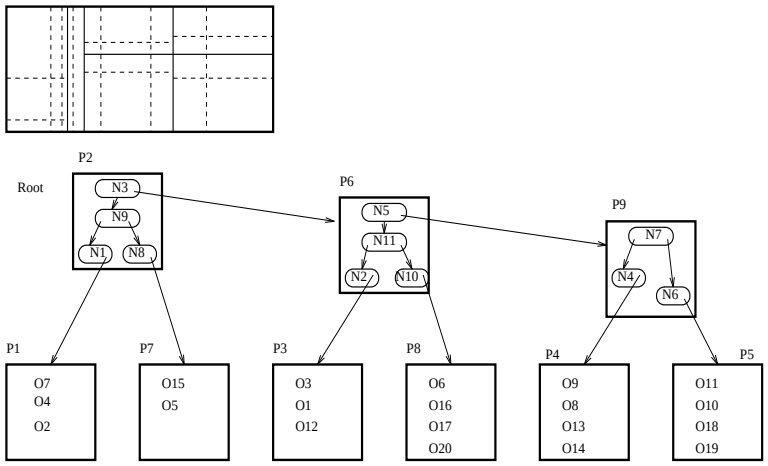
This tree is stored in a page tree which is not a binary tree, as is shown in figure 2.5. Pages contain a page identifier, a page height indicator or level, and the current number of entries in this page. Entries in index pages are both internal and leaf nodes of the sk-d tree. The s2-d nodes are indicated by a node identifier, a node flag to recognise if it is an internal or a leaf node, followed by either the internal node format described above in the case of an internal node or the leaf node format in the case of a leaf node flag. Data pages contain *mbr*s of each object and the object reference. The page occupancy obtained was 79.49%.

The page tree was built following the splitting algorithm included in Ooi's paper. To split a page, the sk-d subtree that approximately splits the page in the middle is chosen. The chosen sk-d subtree is moved to a new page and page entries are updated. Thus, each page begins with an internal node as its root. It is very difficult to obtain a balanced tree in this manner, and paging a binary tree is a subject that requires research.

### 2.2.2 R_tree

This structure is based on the B_tree and it was first presented by A. Guttman in [Gut84] for fast retrieval of data objects by their spatial location. It has similar properties to the B_trees, specifically that the index is completely dynamic, balanced, growing upwards only, and each tuple has a unique identifier. We summarise the B_tree characteristics as follows:

1. the tree is always balanced;
2. worst-case single-object search, insertion (excluding overflow) and deletion (excluding underflow) require no more disk access than the height of the tree. Thus, the access time for a single object is constant and predictable, for a given tree size;




Index pages of the page tree

| Page Id. | Page height | Number of entries | s2-d nodes Internal and leaf nodes are mixed within an index page | |
|---|---|---|---|---|
| P2 | 3 | 4 | N3:F:X:331:345:N9:305:P6 | N9:F:X:306.75:318:N1:282:N8 |
| | | | N1:T:Y:124:175:P1 | N8:T:X:345:282:P7 |
| P6 | 2 | 4 | N5:F:X:423.5:454:N11:418:P9 | N11:F:Y:179:195:N2:147:N10 |
| | | | N2:T:Y:117:195:P3 | N10:T:Y:147:266:P8 |
| P9 | 1 | 3 | N7:F:Y:196:219:N4:172:N6 | N4:T:Y:127:219:P4 |
| | | | N6:T:X:418:540:P5 | |

Data pages of the page tree

| Page Id. | Page height | Number of entries | *mbr* and object reference | |
|---|---|---|---|---|
| P1 | 0 | 3 | 239,139:272,168:O7 | 252,151:282,175:O4 |
| | | | 287,124:318,147:O2 | |
| P3 | 0 | 3 | 326,117:382,188:O3 | 318,124:374,195:O1 |
| | | | 382,166:410,188:O12 | |
| P4 | 0 | 4 | 454,127:481,147:O9 | 481,127:535,194:O8 |
| | | | 449,152:503,219:O13 | 423,152:449,172:O14 |
| P5 | 0 | 4 | 423,172:476,241:O11 | 522,327:540,341:O10 |
| | | | 507,194:535,215:O18 | 418,176:473,244:O19 |
| P7 | 0 | 2 | 282,151:340,225:O15 | 287,147:345,220:O5 |
| P8 | 0 | 4 | 305,147:454,215:O6 | 391,198:444,266:O16 |
| | | | 345,195:374,220:O17 | 391,176:418,198:O20 |

A page identifier is named as P#, a node identifier as N#, and an object identifier as O#.

Figure 2.5: A s2-d tree with 5 entries/index page, and 4 entries/data page.

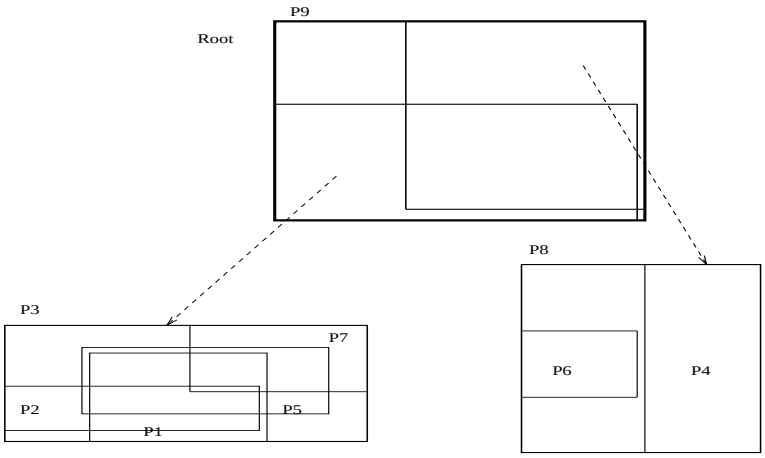
3. overflow and underflow (merge) propagate only upwards in the tree;
4. high average leaf and branch node occupancies, more than 67%;
5. worst-case node occupancy near 50%;
6. the index size is always directly proportional to the quantity of indexed objects;
7. the method is fully dynamic, no reorganizations are required.

The main different property of the R_tree with the B_tree is point 2, where the worst-case single-object search insertion (excluding overflow) and deletion (excluding underflow) require several disk access because overlapping regions.

The leaf nodes contain entries whose format is ($R$, tuple-identifier) where R is the bounding box of the spatial object indexed R=($R_1, \ldots, R_k$) where k is the number of dimensions and $R_j$ is an interval [a,b] describing the extent of the object along dimension j. When an object extends outward indefinitely, either one of the interval boundaries or both may be infinity which is represented by a special character in the tree. Non-leaf node entries have the format: ($R$, child-pointer) where the child-pointer is the address of a lower node in the tree, and $R$ covers all the rectangles in the subtree.

Insertions and splitting are made as in the B_tree. A new entry is inserted in the corresponding leaf node after finding it in the structure. The insertion algorithm calls the splitting algorithm, when the leaf node is full. To split a full node normally the middle of the node is chosen, leaving half of the entries in the original node, and storing the rest of the entries in a new node, which is the brother on the right of the old split node. Deletions are handled by searching the existing entry, and finding the leaf node corresponding to the entry to be deleted. The deletion algorithm contains the call to the merging algorithm which is named *condensed tree* in the original paper. The condensing is made by re-insertion because the conditions for merging nodes are different from the ones of the B_tree due to the spatial properties of the objects included in the *mbr*s. Re-insertion means the remaining entries in a node that underflows must be inserted again in the tree by following the insertion algorithm. Then, the empty node can be deleted.

Finally, this structure uses the technique of overlapping regions in the non-leaf nodes, and it only supports non-zero sized objects. An example is presented in figure 2.6. Each node is stored in a page, that has a page identifier, the actual number of entries, and the page level. Internal pages contain non-leaf node entries and leaf pages store leaf node entries. The percentage of page occupancy obtained




Internal pages of the page tree

| Page Id. | Number of entries | Node height | Internal node entries *mbr* and page reference | |
|---|---|---|---|---|
| P3 | 4 | 1 | 239,124:449,175:P2 | 305,117:454,215:P1 |
| | | | 282,147:503,225:P5 | 391,172:535,244:P7 |
| P8 | 2 | 1 | 345,195:444,266:P6 | 454,127:540,341:P4 |
| P9 | 2 | 2 | 239,117:535,244:P3 | 345,127:540,341:P8 |

Leaf pages of the page tree

| Page Id. | Number of entries | Node height | Leaf node entries *mbr* and object reference | |
|---|---|---|---|---|
| P1 | 3 | 0 | 318,124:374,195:O1 | 326,117:382,188:O3 |
| | | | 305,147:454,215:O6 | |
| P2 | 4 | 0 | 239,139:272,168:O7 | 287,124:318,147:O2 |
| | | | 252,151:282,175:O4 | 423,152:449,172:O14 |
| P4 | 3 | 0 | 481,127:535,194:O8 | 454,127:481,147:O9 |
| | | | 522,327:540,341:O10 | |
| P5 | 4 | 0 | 382,166:410,188:O12 | 287,147:345,220:O5 |
| | | | 449,152:503,219:O13 | 282,151:340,225:O15 |
| P6 | 2 | 0 | 345,195:374,220:O17 | 391,198:444,266:O16 |
| P7 | 4 | 0 | 507,194:535,215:O18 | 423,172:476,241:O11 |
| | | | 418,176:473,244:O19 | 391,176:418,198:O20 |

Figure 2.6: An R_tree with 5 entries/non-leaf page, and 4 entries/leaf page.

in the example of figure 2.6 was of 71.8%. Because objects do not extend outward, there is no need to use the special character mentioned above.

### 2.2.3 $R^+$_tree

The $R^+$_tree structure is an improvement over the R_tree as well as an extension of the K-D-B trees to cover non-zero sized objects. Its main characteristic is to avoid overlapping rectangles in intermediate nodes, and the treatment of underflowing nodes by re-insertion as well. The structure was first presented by T. Sellis, N. Roussopoulos, and C. Faloutsos in [SRF87] for supporting special applications such as CAD/CAM, computer vision and robotics, and expert database systems. It is

a multidimensional index structure based on the storage and retrieval of rectangles saved on pages in secondary storage. The *mbr* is used for handling more complex objects such as circles, polygons, etc.

A leaf node entry of the structure has the following form $(oid, mbr)$, where *oid* is an object identifier, and *mbr* is the minimal bounding rectangle that covers the object identified by *oid*. An intermediate node entry has a similar format which is $(p, mbr)$, where $p$ is a pointer to a lower level node of the tree.

The structure has similar properties to the R_tree and the K-D-B tree in relation to the balance of the tree. It differs from the R_tree in that two entries of an intermediate node cannot overlap. To search and insert an *mbr* in the structure, the $R^+$_tree algorithm uses the same concepts used by B_trees. The main difference is referred to as the split propagation that is made upwards and downwards in the structure. This latter splitting strategy has the disadvantage that it permits tree degeneration which enforces periodic reorganizations of the whole structure.

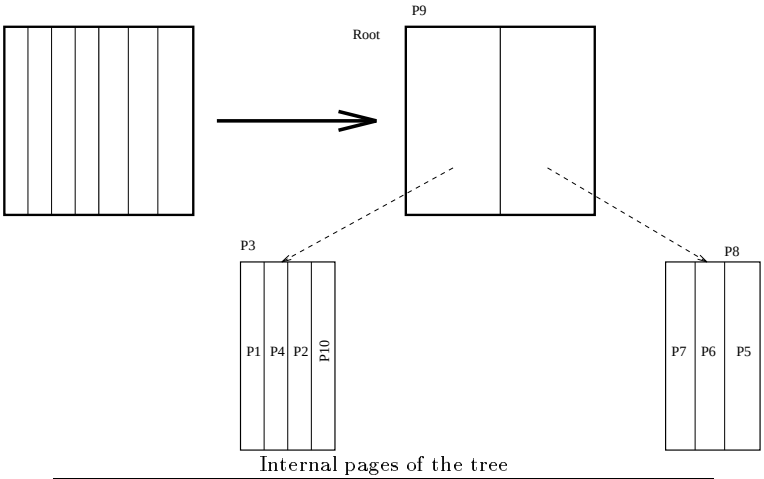
The deletion algorithm follows the same strategy as for the R_tree, but for the $R^+$_tree the deletion of several *mbr*s from leaf nodes is sometimes necessary because the insertion routine may introduce more than one copy for a newly inserted rectangle.

The authors do not give in their paper details about the tree reorganization. They claim that the $R^+$_tree improves search performance compared with the R_tree, especially in the case of point queries. Because the R+_tree does not permit overlapping regions, it starts to split the first overflowed leaf node by including the whole space in the first internal node. This whole space is represented by special characters named $L$ and $H$. $L$ means the lowest value in this dimension and $H$ is the highest value in this dimension.

Figure 2.7 shows an example that was built using the same data as previous structures. Similarly to the R_tree, internal pages contain internal node entries and leaf pages include leaf node entries. A page is identified by a page indicator, and additionally contains the actual number of entries and its height in the tree. The structure resulted in a 67.44 percentage of page occupancy in the example shown.

### 2.2.4 Cell tree

The spatial access method more recently presented in the reviewed bibliography is the cell tree which was created by O. Günther and presented by him in [Gun89]. A cell is a convex set of points which is the representation of the data objects in the




Internal pages of the tree

| Page Id. | Number of entries | Node height | Internal node entries *mbr* and page reference | |
|---|---|---|---|---|
| P3 | 4 | 1 | L,L:287,H:P1 | 287,L:318,H:P4 |
| | | | 318,L:382,H:P2 | 382,L:423,H:P10 |
| P8 | 3 | 1 | 423,L:454,H:P7 | 454,L:481,H:P6 |
| | | | 481,L:H,H:P5 | |
| P9 | 2 | 2 | L,L:423,H:P3 | 423,L:H,H:P8 |

Leaf pages of the tree

| Page Id. | Number of entries | Node height | Leaf node entries *mbr* and object reference | |
|---|---|---|---|---|
| P1 | 2 | 0 | 239,139:272,168:O7 | 252,151:282,175:O4 |
| P2 | 4 | 0 | 287,147:345,220:O5 | 305,147:454,215:O6 |
| | | | 318,124:374,195:O1 | 326,117:382,188:O3 |
| P4 | 3 | 0 | 287,124:318,147:O2 | 287,147:345,220:O5 |
| | | | 305,147:454,215:O6 | |
| P5 | 3 | 0 | 481,127:535,194:O8 | 522,327:540,341:O10 |
| | | | 449,152:503,219:O13 | |
| P6 | 3 | 0 | 454,127:481,147:O9 | 423,172:476,241:O11 |
| | | | 449,152:503,219:O13 | |
| P7 | 3 | 0 | 423,172:476,241:O11 | 449,152:503,219:O13 |
| | | | 423,152:449,172:O14 | |
| P10 | 2 | 0 | 305,147:454,215:O6 | 382,166:410,188:O12 |

Figure 2.7: An $R^+$_tree with 5 entries/intermediate page, and 4 entries/leaf page.

tree. Thus, the cell tree uses cells instead of *mbr*s. This is a balanced tree structure whose leaves contain cells, and whose interior nodes contain convex polyhedra. In the author's first paper, he presented the cell tree as 'an object-oriented dynamic index structure for geometric databases'. Later on, in another article [GB91], he referred to the structure as 'a new dynamic access method for spatial databases'. It seems to us that both affirmations are right, and we take the cell tree as a SAM.

The nonstandard database applications where the cell tree can be used are similar to those of the R$^+$_tree. It is based on both the binary space partitioning (BSP) tree, which is a binary tree representing a recursive subdivision of a given space into subspaces by means of (k-1)-dimensional hyperplanes, and on the R_tree. The cells are indexed based on its location in space. The tree is dynamic and it does not need periodic reorganization.

A leaf node entry represents a cell by a tuple, $(G, D, A)$, where $G$ is the geometry of the cell including the identifier, $D$ is the data object represented by a convex chain, and $A$ is a set of all attributes of $D$ that may be required to answer a query in the given application. An interior node contains entries of the form $(cp, P, C)$ where $cp$ is the address of the descendant node, $P$ is a convex, not necessarily bounded, k-dimensional polyhedron called a partition, and $C$, named the *container*, is a convex subset of $P$, which also contains each cell in the subtree.

It is a balanced tree, each tree node corresponds to exactly one disk page, and has an order $m$ for specifying the minimum number of entries in an interior node. The clustering of the data objects depending on their location in space is an intrinsic property of the cell tree. The use of the clipping technique permits the partitioning of space into regions that do not overlap.

Insertions are made in a similar manner to B_trees because new cells are added to the leaf nodes, that are split when they overflow. The splitting strategy only propagates up the tree, but sometimes it is possible that there is no hyperplane that splits a leaf node. In this case, the leaf node is stored using overflow nodes. Deletions are treated like deletions in a B_tree as well. Condensing the tree eliminates empty leaves and it only propagates up the tree. A cell tree with $m = 1$ where partitions and containers are *mbr* is a special case of the R$^+$_tree.

Pages are identified by a page indicator, the current number of entries and the page height. Additionally, leaf pages contain a page indicator to allow for overflow pages, which are treated here as a single linked list of pages.

Figure 2.8 presents an example where *mbr*s are used as containers, m=2, and partitions are made in rectangular shapes. It can be seen that the cell tree fragments

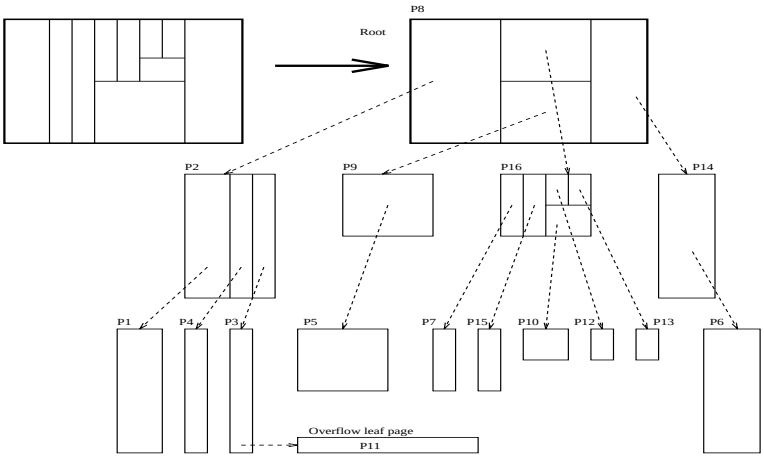
objects. For example, the figure shows that object 6 is fragmented eight times, so the O6 reference appears in six different leaf pages. It is important to highlight that page 3 has an overflow page identified by page eleven. This situation is attained because the splitting algorithm fails to find a hyperplane to split the page. The author claims that this situation seldom arises. The page occupancy reached by the example is 73.91%.

## 2.3 Properties of comparison and the comparative table

Before presenting the comparative table between MPAM and SAM, we briefly describe each property of the table. Comments included in the table are taken from each author's paper for each structure. When a concept is not present in the reviewed article, the message "not referenced" is written in the appropriate column.

In table 2.1, under the property named **general characteristics**, we mention the most relevant property of the structure, generally, based on other known structures as B_trees or hash tables. **Nodes** mainly describe the types of nodes supported by the hierarchical structure when relevant. These types are better shown in figures, but we omit these figures here because we do not consider this level of detail. The **order** is an important property because the majority of the proposed structures follow the B_tree's principles, considered by people as the best properties that an access method should have [Fre93]. That is the reason why the majority of the structures are based on the B_tree. The order defines the minimal and maximal number of entries in a node. It is important to obtain node occupancy in average and in worst-cases. An average node occupancy of more than 65% is considered good.

In table 2.2, the next property is **spatial object**. It basically indicates the shape used for treating the spatial object indexing. Rectilinear means the use of bounding boxes associated with regions and spatial objects themselves. The majority of the structures use *mbr*s parallel to the X-Y axes because this simplest shape of a possible object container, but other shapes may be used as well. The use of *mbr*s leads to minimum storage space, but precludes specific object retrieval because many objects can fit in the same *mbr*. On the other hand, the use of a more complicated container shape needs more space in each node because entries are longer, but increases the precision of retrievals because the container shape is near the object shape.




Internal pages of the tree

| Page Id. | Number of entries | Node height | Page reference and hyperplane |
|---|---|---|---|
| P2 | 3 | 1 | P1:L,L:287,H:239,139:287,225 |
| | | | P4:287,L:318,H:287,124:318,225 |
| | | | P3:318,L:382,H:318,177:382,225 |
| P8 | 4 | 2 | P2:L,L:382,H:239,117:382,225 |
| | | | P9:382,L:481,152:382,127:481,152 |
| | | | P16:382,152:481,H:382,152:418,266 |
| | | | P14:481,L:H,H:418,127:540,341 |
| P9 | 1 | 1 | P5:382,L:481,152:382,127:481,152 |
| P14 | 1 | 1 | P6:481,L:H,H:481,127:540,341 |
| P16 | 5 | 1 | P7:382,152:418,H:382,152:418,266 |
| | | | P15:418,152:423,H:418,152:423,266 |
| | | | P10:423,152:481,172:423,152:481,172 |
| | | | P12:423,172:449,H:423,172:449,266 |
| | | | P13:449,172:481,H:449,172:481,244 |

Leaf pages of the tree

| Page Id. | Overflow page Id. | Number of entries | Node height | Leaf node entries *mbr* and object reference | |
|---|---|---|---|---|---|
| P1 | 0 | 3 | 0 | 239,139:272,168:O7 | 252,151:282,175:O4 |
| | | | | 282,151:287,225:O15 | |
| P3 | P11 | 4 | 0 | 318,147:345,220:O5 | 318,124:374,195:O1 |
| | | | | 318,147:383,215:O6 | 326,117:382,188:O3 |
| P4 | 0 | 4 | 0 | 287,124:318,147:O2 | 287,147:318,220:O5 |
| | | | | 305,147:318,215:O6 | 287,151:318,225:O15 |
| P5 | 0 | 2 | 0 | 454,127:481,147:O9 | 382,147:454,152:O6 |
| P6 | 0 | 4 | 0 | 481,127:535,194:O8 | 522,327:540,341:O10 |
| | | | | 481,152:503,219:O13 | 507,194:535,215:O18 |
| P7 | 0 | 4 | 0 | 382,152:418,215:O6 | 382,166:410,188:O12 |
| | | | | 391,198:418,266:O16 | 391,176:418,198:O20 |
| P10 | 0 | 3 | 0 | 423,152:454,172:O6 | 423,152:449,172:O14 |
| | | | | 449,152:481,172:O13 | |
| P11 | 0 | 2 | 0 | 318,151:340,225:O15 | 345,195:374,220:O17 |
| P12 | 0 | 4 | 0 | 423,172:449,215:O6 | 423,172:449,241:O11 |
| | | | | 423,176:449,244:O19 | 423,198:444,266:O16 |
| P13 | 0 | 4 | 0 | 449,172:454,215:O6 | 449,172:476,241:O11 |
| | | | | 449,176:473,244:O19 | 449,172:481,219:O13 |
| P15 | 0 | 3 | 0 | 418,152:423,215:O6 | 418,198:423,266:O16 |
| | | | | 418,176:423,244:O19 | |

Figure 2.8: A cell tree with 5 entries/interior page, and 4 entries/leaf page.

| *structures* | *Properties* | | |
|---|---|---|---|
| | *General characteristics* | *Nodes* | *Order* |
| *K-D-B tree* | Combination of B_tree and k-d tree. | Similar to the B_tree nodes. | m: upper and lower bound as in B_tree. |
| *Grid file* | Adaptable and symmetric multikey file structure based on hash tables. | It is not a tree. The data buckets contain objects in the same grid directory. | The concept is not present. |
| *buddy tree* | Combination of grid file and R_tree. Avoid partitioning empty data space. | Each directory node contains a collection of *mbr*. | m: upper and lower bound as in B_tree. |
| *sk-d tree* | Generalisation of k-d tree to spatial case. It is not balanced. | Two types of nodes, binary tree nodes and page tree nodes. | The concept is not present. |
| *R_tree* | Generalisation of B_tree to higher dimensions. | Each node corresponds to a bounding box. Leaves contain the objects covered by a *mbr*. | m: upper and lower bound for the number of descendants of an interior node. m:lower,M:upper $m \leq \frac{M}{2}$ |
| $R^+$*_tree* | Improvement of R_tree with respect to search operators. | See R_tree. | The concept is not present. |
| *Cell tree* | Combination of BSP_tree and $R^+$_tree. Data object is represented as unions of convex point sets (cells). | Each node corresponds to a cell. Leaves contain the cells whose interior nodes correspond to a hierarchy of nested convex polyhedra. | m: minimum number of entries in an interior node. |

Table 2.1: Comparative table. Part 1.

The **spatial technique** shows the name of the known techniques used to extend a point indexing structure to a multidimensional one. These techniques are transformation, clipping, and overlapping regions. The *transformation* technique treats objects in a d-dimensional space as points in a 2-dimensional space. Thus, a bidimensional rectangle described by $(X_1, Y_1, X_2, Y_2)$ is represented as a point in 4D space. The second technique, *clipping*, partitions the d-dimensional data space into pairwise disjoint subspaces. Thus, an object is partitioned into several disjoint smaller objects so each smaller object is totally included in a subspace. Finally, *overlapping regions* permit overlapping subspaces such that objects are totally included in one of the subspaces. The first technique has the disadvantage that the transformation must be chosen to preserve the spatial properties, and the third one presents the problem of multiple path searches for retrieving a needed object.

The property named **insertion and deletion of data objects** summarizes briefly how an entry insertion or deletion is made in the structure, and which level of the structure is affected. In hierarchical structures, the new entry is normally inserted in a leaf page, but to choose the appropriate leaf page, a descending path search has to be done. Similarly, to delete an existing entry, the descending path search is made to choose the leaf node containing the entry to be deleted. In non-hierarchical structures, the place where the new entry must be inserted or where the existing entry is lying, is chosen by using a function that transforms the key value normally into an address value. This last method used in non-hierarchical structures is better than the former one utilised in hierarchical structures, when the value obtained after a transformation is unique for each key value in the application.

Table 2.3 shows the **position of the splitting hyperplane** that can be *adaptable*, when the position of the splitting either may be chosen according to conditions on data values, or *fixed* when whatever values are present in the structure, the point of splitting is always the same. The advantage of the first of these splitting strategies is that it tends to lead to a more compact index structure. The **dimensionality** which can be in only 1-dimension or in all d-dimensions, indicates that the splitting position may be put in one of the defined dimension at a particular time, or in the defined d-dimension as whole. In the case of choosing 1-dimension, this selection may be made cyclically by following a sequence of dimensions, or acyclically by selecting the dimension based on some defined selection policy. The **locality of the hyperplane** can affect all the regions in the given direction (called global), or solely inside the region to be split (named local). The local strategy is generally preferred because the effect of the splitting policy is restricted to the overflowed node.

| | *Properties* | | |
|---|---|---|---|
| *structures* | *Spatial data object* | *Technique* | *Insertion and deletion of data objects* |
| *K-D-B tree* | Rectilinear. | Not referenced. | Combination of the B_tree and k-d tree algorithms. |
| *Grid file* | Rectilinear. | Transformation. | Insert/delete an entry in a data bucket. |
| *buddy tree* | Rectilinear. | It can be implemented with either clipping, overlapping regions, or transformation. | Reorganization restricted to nodes in the search path. |
| *sk-d tree* | Rectilinear. | Overlapping regions. | As in the k-d tree. |
| *R_tree* | Rectilinear. | Overlapping regions. | Insertion/deletion involves one leaf entry. |
| *$R^+$_tree* | Rectilinear. | Clipping. | Insertion/deletion may involve more than one leaf entry. |
| *Cell tree* | Not rectilinear. | Clipping. | Insertion/deletion involve several leaves. |

Table 2.2: Comparative table. Part 2.

| | *Properties* | | |
|---|---|---|---|
| *structures* | *Position of the splitting hyperplane* | *dimensionality* | *Locality of the hyperplane* |
| *K-D-B tree* | Adaptable. (Not fixed) | One dimension chosen cyclically. | Local, called brickwall methods. |
| *Grid file* | Fixed. Always by halving. | One dimension. | Global, called grid methods. |
| *buddy tree* | Fixed. | One dimension. | Local. |
| *sk-d tree* | Adaptable. | One dimension chosen cyclically. | Local. |
| *R_tree* | Adaptable. | d-dimensional. | Local. |
| *$R^+$_tree* | Adaptable. | d-dimensional. | Local. |
| *Cell tree* | Adaptable. | d-dimensional. | Local. |

Table 2.3: Comparative table. Part 3.

In table 2.4, the **binary division** indicates how the division into the nodes of the structure is made. A division is balanced if it divides the region into subregions with a similar number of objects into each one. This division can be based on either a criterion for choosing the dimension of splitting or a criterion for positioning the partition boundary or splitting hyper-plane.

The **splits/merges propagation** presents the direction followed when one of the splitting/merging operations occurs, and in some cases, the number of nodes involved in these operations. It is considered a good direction if only the father nodes in the path from the root to a leaf node are involved, in the case of hierarchical structures. For non-hierarchical structures, the propagation should only involve the structure placed between the bucket containing the object and the structure used to transform the key value. A propagation up and down a tree is normally considered to be a poor propagation strategy.

**Clustering** describes if the structure imposes object clustering on disk pages, and if so, how it does it. In this case, we are interested in clustering the objects by proximity because the indexed objects are all spatial objects. Those structures that do not cluster objects by proximity are generally considered no good for indexing spatial objects. There are two cases where this point is not clear, so we mention them as not referenced. Intrinsic means that the clustering is imposed by the indexing method.

Another possible classification is showed by Frank and Barrera in [FB89], and presented in table 2.5. The **type of geometric data** indicates if the access method refers to isolated points or regions, so it defines if the method is a MPAM or a SAM. To index spatial objects, SAMs are preferred over MPAMs. The **handling of objects** can be cataloged as fragmenting, if the access method divides the object assigning each fragment to a unique page, or non-fragmenting, if the structure maintains the integrity of the objects and performs extra disk accesses. One of the special requirements to index spatial objects is the non fragmentation of them, because they are not spatially decomposable. As mentioned in [Fra91],

> ... objects are non-atomics (i.e., complex in the object-oriented sense) but non spatially decomposable (in spatial pieces that still have the same meaning).

The **retrieval method** takes into account the implementation of the mapping between spatial and disk spaces by following either a tree (hierarchical methods) or a function (hashing methods). Both are normally seen as good access methods.

| | *Properties* | | |
|---|---|---|---|
| *structures* | *Binary division* | *Splits/Merges propagation* | *Clustering* |
| *K-D-B tree* | Balanced. Dimension is fixed according to node dimension. | Only up the tree. | Based on the lexicographically order. Not by proximity. |
| *Grid file* | Balanced according to a defined binary radix interval | Up the directory and the scales. | Based on proximity. |
| *buddy tree* | Balanced and regular (represented by a k-d trie) | Only up the tree. | Not referenced. |
| *sk-d tree* | Balanced. Choosing the dimension of the longest side of the subspace. | Down the binary tree and up the page tree. | Not referenced. |
| *R_tree* | Strict. The partition boundary always lies halfway along the partition interval. | Only up the tree. | Not necessarily. Worst case (no clustering): one disk access for each data object retrieved. Best case: all data object whose data intervals are stored in the same R or $R^+$_tree node. |
| *$R^+$_tree* | Balanced. See cell tree. | Up and down the tree. | See R_tree. |
| *Cell tree* | Balanced. The partition boundary can be positioned anywhere, according to arbitrary balancing criteria. | Only up the tree. | Intrinsic. Each leaf node corresponds to a convex partition in space and all the cells in that partition are actually stored on the corresponding disk pages. |

Table 2.4: Comparative table. Part 4.

| *structures* | *Properties* | | |
|---|---|---|---|
| | *Type of geometric data* | *Handling of objects* | *Retrieval method* |
| *K-D-B tree* | Point-oriented | non-fragmenting | hierarchical |
| *Grid file* | Point-oriented | non-fragmenting | direct |
| *buddy tree* | Point-oriented | fragmenting | hierarchical |
| *sk-d tree* | Region-oriented | non-fragmenting | hierarchical |
| *R_tree* | Region-oriented | non-fragmenting | hierarchical |
| *$R^+$_tree* | Region-oriented | fragmenting | hierarchical |
| *Cell tree* | Region-oriented | fragmenting | hierarchical |

Table 2.5: Comparative table. Part 5.

Finally, table 2.6 presents the last three properties. **Periodic reorganization** indicates if the structure needs reorganization of its nodes due to the splitting/merging strategies. The majority of the structures need not be reorganized after an insertion or deletion, and this property is considered very good for an access method, because that means a structure which preserves its properties among entries insertions or deletions. The column labeled **advantages** shows the main advantages presented by the structure highlighted by either its author or by others researchers. In a similar manner, **disadvantages** presents the main disadvantages referenced by researchers in the area.

## 2.4 Spatio-temporal access methods

These access methods are presented outside the comparison table because they are not comparable with the previous ones. We include two access methods that are proposed specifically to deal with spatio-temporal data objects. A spatio- temporal data object is an object that may change its location and/or shape at different time intervals. We only include here the RT_tree presented by X. Xu et al. in [XHL90] and the TR*_tree proposed by R. Schneider and H. -P. Kriegel in [SK92].

### 2.4.1 The RT_tree

This structure is proposed to deal with a collection of entities contained in images and representing spatio-temporal data objects, which have spatial and temporal attributes. An RT_tree of order **M** is a height balanced tree with the index data objects in its leaf nodes containing entries (MBR, T, R), where the MBR is the minimal bounding rectangle that completely covers the data object identifying its

| | *Properties* | | |
|---|---|---|---|
| *structures* | *Periodic reorganization* | *Advantages* | *Disadvantages* |
| *K-D-B tree* | Not required. | Least complicated of the MPAM. | No cluster by proximity. |
| *Grid file* | Not required. | Only two disk access in exact match queries. | The directory does not expand at the same rate as the data. |
| *buddy tree* | Not required. | Avoid the empty space partitioning. It does not depend on the order of data insertion. The best performances in relation to others MPAM and the R_tree. | It is not balanced. |
| *sk-d tree* | Not required. | Efficient searching in case of intersections. | Complicated splitting strategy of the page tree. |
| *R_tree* | Not required. | Least complicated of the SAM. | Object intersection introduces an ambiguity in its location. Several search paths are traversed in point search. Fail in partitioning nested object covers. Updating by reinsertion causing possible deadlock. |
| $R^+$*_tree* | It is required because the structure can degenerate. | Fast computation of search operators. Guarantees a shortest path search. | Possible degeneration of the structure. Storage utilization may deteriorate. Inevitable overlapping. Traversal of unpredictable length during insertion and deletion. Nested object covers are not supported. Fragmentation. |
| *Cell tree* | Not required. | Less disk accesses than the others. Point and range searches are efficient. Guarantees a shortest path search. | More storage space and CPU time, use overflow records in case of nested object covers, traversals of unpredictable length during insertion and deletion. Fragmentation. |

Table 2.6: Comparative table. Part 6.

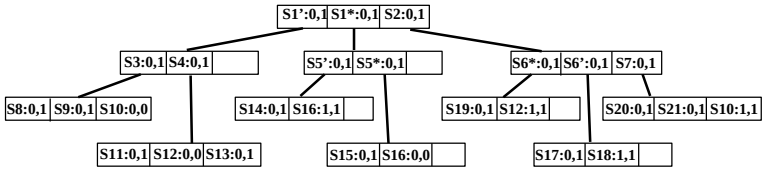



Figure 2.9: The RT_tree. Example taken from X. Xu et al. reference.

spatial location and/or shape, T indicates the time interval from time Ti to Ts which is the interval when the data object is at MBR, and R is the object reference in the database. The nonleaf nodes contain entries of the same format, but the MBR is calculated to cover all the MBR in its child node, and R is a reference to its child node. All the leaf nodes are in the same level, the last one, and they are chained together to permit sequential search. This structure allows MBRs to overlap in the nonleaf nodes. Figure 2.9 presents an example of this structure. We do not present the example by using the data of the last structures because this data does not contain temporal information.

The insertion process is similar to the R_tree for the first image at a particular time interval. When a new image comes at a different time interval, the insertion process searches first the leaf node where the new entry is to be placed by checking whether there is an entry with the same MBR and the same data. If there is one, the time interval is expanded to contain the new entry; otherwise, checks are made as to whether this node has enough room for a new entry. If not, the leaf node is split following a splitting policy, otherwise the new entry is inserted there. The deletion process is not mentioned in the paper, but it considers a reorganization process to maintain the tree. The splitting strategy can be either based on a spatial coverage minimizing the area of the MBR, or a time interval including overlapping time intervals within the child node, or a semantic coverage based on the semantic knowledge about the images. The first splitting strategy is normally preferred. It treats spatial and temporal data separately, and no symmetrically favouring spatial processing over the temporal one. This is basically, a spatial structure extended to support time intervals.

### 2.4.2 The TR*_tree

To index polygonal objects, R. Schneider and H. -P. Kriegel in [SK91] have proposed use of the decomposition approach, which handles complex polygonal objects by decomposing them into a set of simple components as trapezoids by using the plane sweep technique. This technique sends out for each vertex of a polygon, one or

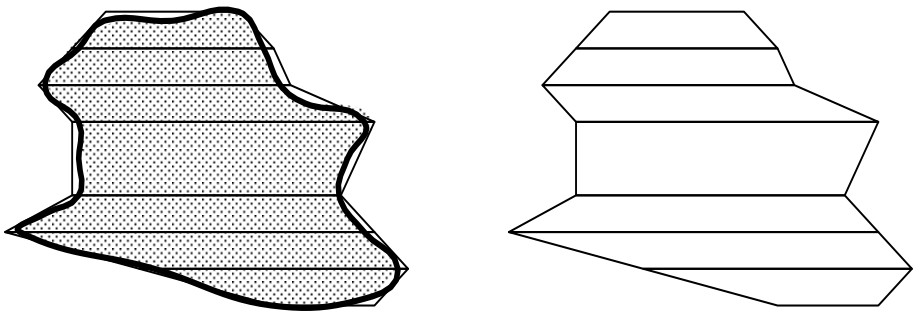


Figure 2.10: Example of the plane sweep technique.

two rays into the interior of the polygon to the first edge encountered, as shown in figure 2.10. The set of trapezoids is indexed with an TR*_tree, which is derived from the R_tree. Each leaf node contains a set of entries of the form **(Oid, trapezoid)**, where **Oid** is the object identifier assigned to the trapezoid, and **trapezoid** is a component of the decomposed polygon identified by Oid. A non leaf node contains a set of non leaf entries having the same format as the R_tree, that is **(R,mbr)** where **R** is the reference of the child node, and **mbr** is the minimum bounding rectangle of all rectangles in that child node.

This structure is extended by the authors in [SK92] to contain temporal information as well as spatial. Thus, a time stamp **ti** is added to the leaf entries, which now have the following form **(Oid, trapezoid, ti)**. To search an object, first a spatial search is performed to find all the trapezoids that answer the spatial condition, and secondly a temporal search is done over this set of trapezoids, to select a subset of them fulfilling the temporal conditions.

Summarizing, this structure is similar to the R_tree in the non leaf nodes, and changes the format of the leaf nodes to manage a set of trapezoids representing decomposed polygonal objects. Because of the mentioned similarity, we can use figure 2.9 that represents such a tree referred to the internal pages, and changing the time intervals for time stamps. The leaf pages change its format to contain each one of the trapezoids of the decomposed objects, and putting time stamps instead of time intervals. As the previous structure, it is basically a spatial index extended to support time stamps associated with the spatial information.

## Summary

In general, the position of the object in space is the main key in retrieval operations. Thus, SAMs are the more recently proposed access methods which emphasise the use

of the location in space as the more important attribute in indexing objects. SAMs attack the root problem of spatial objects indexing instead of extending a PAM with spatial retrieval capacities. A large number of index structures have been proposed for handling multidimensional point and spatial data. Quadtrees and octrees are mainly used for raster representation and they are not included here.

Following the comparative table, the R_tree can be viewed as the simplest SAM that can be improved by changing search operators and supporting nested object covers by overflow leaf pages. In relation to the simplicity of the index structure, the K-D-B tree is the simplest and cell tree is the more complicated. R_tree and K-D-B trees are especially good because they maintain many advantages of B_trees such as balancing and dynamicity, but the K-D-B tree does not cluster objects by proximity. The cell tree can be very efficient because it stores cells of arbitrary shape instead of *mbr*s and the rest of the attributes in its nodes without having the data objects stored in a file apart from the index, but fragmentation and use of overflow leaf pages may well deteriorate this efficiency.

The grid file structure handles efficiently a collection of d-dimensional data records (d<10) where each dimension has a large and linearly ordered domain of values. It supports access for each dimension separately, for each combination of these dimensions taken from two, three, four, etc until d-dimensions, which means that all the previous indexes are packed in this structure. Its main disadvantage is that the directory grows large when the data distribution is non-uniform, thus its use is highly recommended only for uniform data distribution.

The rest of the structures, excepting the sk-d tree, have simpler concepts and they permit less complicated implementations. Among the better MPAMs, the buddy tree properties may be seen as better than others, emphasizing the avoidance of empty space partitioning, and the possibility of the use of any of the spatial techniques (clipping, overlapping, or transformation).

In summary, structures based on grid files can be used to retrieve objects having multiple keys, where each key value is a point. If distribution of values is uniform then the scale-based grid file is recommended, otherwise the buddy tree is good. For fast implementation, the K-D-B tree is recommended for point data and R_tree for spatial data. If fast retrieval is needed because the number of disk accesses is a crucial factor, then R+_tree is recommended using *mbr*s or the cell tree may be adopted by using object containers of an arbitrary shape. To have the possibility of retrieving using multiple keys separately, the scale-based grid files are preferred. If a low level of maintenance is needed, all the structures can be chosen except

R+_tree, due to possible structure degeneration. For avoiding fragmentation, it is preferable to avoid grid file, R+_tree, and cell tree. To have a structure designed for paged memory, it is not recommended to choose those based on binary trees, like sk-d trees. Finally, if data objects contain nested object covers, it is possible to take buddy trees that can manage this type of objects without overflow pages. If overflow pages do not matter, then sk-d trees or the cell tree can be used.

To handle spatio-temporal data objects, it is possible to use any of the mentioned structures, but it is preferred to choose those that may easily be extended to include temporal time intervals. In such a case, the RT_tree is preferred due to the simplicity of its concepts. In the case of having complex polygonal objects, the solution proposed by Schneider and Kriegel violates the requirements exposed by Frank in [Fra91] referring to the non decomposition of spatial data objects, but it is a very interesting proposition. Furthermore, these two index structures support temporal data but favouring spatial processing over the temporal one. We are interested in an access method that may treat both types of data symmetrically, and that can favour one over the other when the application requires it. For reaching that goal, we consider that the structure must be created based on both types of data, and also permits separated searches under different application considerations. Thus, we choose the R_tree where the R_tree nodes will contain d-dimensional rectangles (hyper-rectangles), and searches will be supported by adding either d-dimensional, spatial, and temporal operators.

# Chapter 3

# Spatio-temporal concepts in OODB

A spatio-temporal object is an abstraction of an entity having an identifier, a spatial location and shape, a temporal property, and some other characteristics that describe it. This chapter deals with the description of this type of objects by presenting a brief summary of the main concepts in object-oriented databases. The main purpose of this chapter is to define object-oriented general concepts, and introduce spatial, temporal, and spatio-temporal databases, including SQL extensions. The chapter is organised as follows: Section 3.1 presents object-oriented objects, their relationships, and their utilization in object-oriented databases (OODB). Section 3.2 describes spatial objects and spatial databases that support the use of these objects. Section 3.3 deals with temporal objects supported by temporal databases. Section 3.4 contains the main concepts to index and query a spatio-temporal database. Finally, section 3.5 summarizes a way ahead for indexing spatio-temporal objects.

## 3.1 Object-oriented objects

The object-oriented approach is based on the concept of ontology [Bun77], where

**i)** the world is composed of things (entities)

**ii)** the forms are properties of things

**iii)** things are grouped in systems

**iv)** each system, except the universe, interacts with other systems in at least one way

**v)** each thing changes and obeys laws.

An object, in the Ontologic sense, is any thing that can be known or represented by a subject. An entity is a concrete object that has properties and behaviour, represented by the state and the change in the state of its properties, respectively. In this theory, the real world is an aggregation of objects that

**a.** have properties (attributes)

**b.** have relationships with one other

**c.** obey laws

**d.** have state which is transformable based on events

**e.** change.

In Computer Science, the object-oriented approach captures a portion of the real world named an application domain, in terms of entities and treats these entities or objects based on the concepts of encapsulation and extensibility. Encapsulation provides a form of logical data independence by encapsulating both data (structure) and programs (behaviour) of the object, and only permitting access to the object throughout a defined object public interface, where is indicated their methods that can be invoked by message passing. Whilst extensibility refers to the ability to extend an existing object by adding structure through inheritance and/or behaviour to the original object interface. Extensibility can be reached in two ways: behavioural extension and inheritance. The first way is supported by adding more programs to the object, and the second one, by including new *is-a* relationships which are described in the next subsection.

### 3.1.1 Object-oriented relationships

These are *is-a* and *part-of* relationships. The first one describes an object class which *is-a* specialization of some other object class. The second one expresses an object composed by other objects. According to Hughes in [Hug91], these two relationships are supported by five abstraction concepts, such as: classification, identification, aggregation, generalization, and specialization. *Classification* groups objects

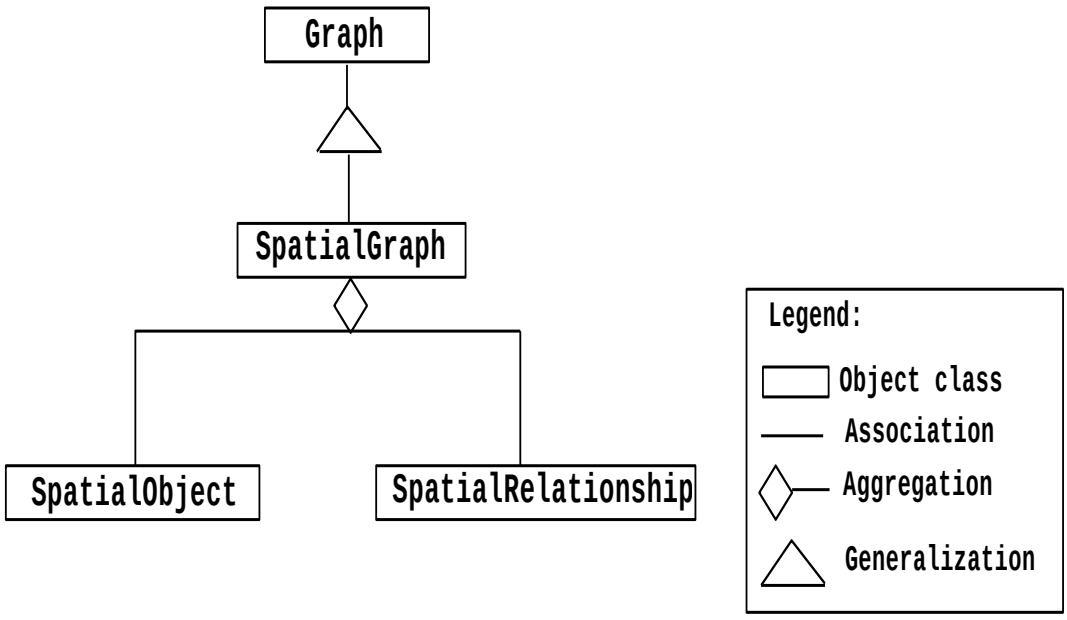



Figure 3.1: Object relationships.

with similar properties and behaviour into object types or classes. *Identification* is supported by an identifier associated to each object that identifies uniquely each of them. *Aggregation* or composition represents the part-of relationship among objects by a higher level aggregate object. *Generalization* represents a set of objects with similar properties by a generic object. *Specialization* describes a set of objects with similar characteristics by a particularised object. Both, generalization and specialization support the is-a relationship, and object inheritance. Figure 3.1 illustrates these two relationships among objects. Each spatial graph is a graph which is composed of spatial objects, and spatial relationships. **Graph** class is a generalization of the **SpatialGraph** class.

### 3.1.2 Object-oriented databases

An object-oriented database (OODB) is a database that integrates essential concepts from object-orientation with capabilities of databases. The main concepts that have to be supported by OODBs have been specified by M. Atkinson et al. in [ABD+89], namely:

object identity (identification),

types or classes (classification),

inheritance (generalization/specialization),

**Basic classes**

| **Magnitude classes** | **Collection** | **Graphical** |
|---|---|---|
| Character | ArrayOf[X] | Form (Image) |
| Date | String / Text | Point |
| Time | SetOf[X] | Rectangle |
| Integer | ListOf[X] | Paragraph |
| Float | Dictionary[X,Y] | |
| Association | OrderedCollectionOf[X] | |

**D/K multimedia classes**

| **Textual** | **Graphical** | **Image** | **Hypermedia** |
|---|---|---|---|
| Text | GraphicalPoint | Image | Unit |
| Paragraph | Line | | Link |
| | Polygon | | Button |
| | Window | | HyperNetwork |

Figure 3.2: Basic and multimedia classes of the D/K model.

complex objects which are built from simpler ones by applying constructors to them, such as: tuples, sets, arrays, bags, lists;

encapsulation that provides a form of data independence by treating structure and behaviour of the objects by using their operations;

overriding that permits the redefinition of operations resulting in a single name denoting a set of different programs; and

late binding that solves the problem of identifying which program with the same name is needed at run-time.

To the best of our knowledge, there is not a unique and accepted object model for representing object-oriented databases as in relational ones. Because of this, W. Kim in [Kim90] defines OODB in relation to an object-oriented data model, suggesting that this data model is defined before the system is designed.

In this work, we especially consider the D/K data model proposed by J. Montilva in [Mon93], because one of the purposes is the definition of an indexing method that could be used for supporting recovery and secondary storage management. This model is an integration of some OODB, multimedia, and knowledge models. Figure 3.2 shows the basic and D/K multimedia classes of this model already explained and defined by Montilva in his PhD thesis.

To support our indexing method, defined in the next chapter, three basic classes that extend the D/K model basic classes are presented in figure 3.3. The *HR* class refers to the hyper-rectangles used in the definition of the index entries. A hyper-rectangle is a finite set of intervals, one for each defined dimension (DIM) in the

| Basic classes | | |
|---|---|---|
| **Magnitude classes** | **Collection** | **Graphical** |
| Character | ArrayOf[X] | Form (Image) |
| Date | String / Text | MPoint |
| Time | SetOf[X] | HR |
| Integer | ListOf[X] | Paragraph |
| Float | Dictionary[X,Y] | |
| Association | OrderedCollectionOf[X] | |
| | Interval | |


Figure 3.3: Extension to the basic classes of the D/K model.

tree. This class generalises the Rectangle class of the D/K model. The *Interval* class is a pair of real numbers that defines the minimum and maximum values of the object interval. Finally, the *MPoint* is a multi-dimensional point composed of $d$ real numbers, one for each of the d dimensions already defined. MPoint is a generalization to many dimensions of the Point class of the original model.

Querying an OODB depends on the data model used in the system. There are two main attempts to standardise object-oriented models. The first one corresponds to the Object Model Group (OMG), presented by W. Kim in [Kim95]. This extension is a superset of the *structured query language* (SQL), which uses keywords added to the SQL commands to describe an object-oriented query. The second one is called the ODMG-93 group that presents its standard model in [ADF+94]. Both groups use a different extension of the standard commercial relational database language named SQL. Querying command in SQL is as follows

*select* listOfAttributes

*from* declarationOfRelations

*where* predicates

*groupBy* groupingSpecification

*orderBy* orderingSpecification

The SQL extension of the OMG is named OSQL, and that of the ODMG-93 is called OQL, each one is detailed in the given references. Both extensions are

declarative languages as well as SQL, but the structures that they manipulate are different from (and generally more complex) than the structures (i.e. tables) that SQL manipulates. They have a SQL syntax including some special operators and keywords to deal with object-oriented concepts not supported by SQL standard. As examples of these operators and keywords, we find in OSQL the keyword *atomic* meaning a bag of values, and *distinct* indicating a set of values, both are optionally set in the select clause. In OQL, the keyword *distinct* has the same meaning as in OSQL, and additionally, we find the keyword *struct* referring to the construction of a tuple of attributes. In particular, OQL supports object identity, complex structured objects (sets, bags, lists, etc), subtype and supertype relationships, and the ability to access operations (methods) as well as attributes by considering all of them (even the query itself) as expressions. This last point, the ability to invoke object methods, is particularly pertinent here because it permits the use of spatio-temporal operators to deal with spatio-temporal relationships, which will be defined later in this chapter.

### 3.1.3 Indexing object-oriented objects

The main proposed and commercial OODBMSs use $B^+$_trees as the indexing method, as described in the following references [MS90], [WK90], [Kim90], [Deu90], [MS86], [BK89].

The work that studies this problem extensively is that of E. Bertino and W. Kim in [BK89], where it is shown that $B^+$_trees can be used for the three types of indexes considered in ORION system. A *class-hierarchy* index provides access to all objects within a class hierarchy via some attribute, *at1*, say. The indexed attribute (at1) will be inherited downwards at the class level by virtue of the specialisation hierarchy so that all object instances will have a value for this attribute. Figure 3.4 illustrates a class-hierarchy index rooted at class A which has two subclasses. A *nested-attribute* index is maintained on a class-composition hierarchy. The indexed attribute, *at2* say, is inherited upwards at the object level by virtue of composition. Each object instance will have a value for this attribute either directly or through one of its components. A nested-attribute index on a composed attribute *at2* is shown in figure 3.5. Remembering that *at2* is referenced by composition, so it is also an attribute of *class A* and *class B*. Finally, the *two-dimensional* index is a nested-attribute index augmented with a class-hierarchy index for each class in the sequence of classes between the class and the class to which the indexed attribute belongs. This situation is presented in figure 3.6. E. Bertino and W. Kim have

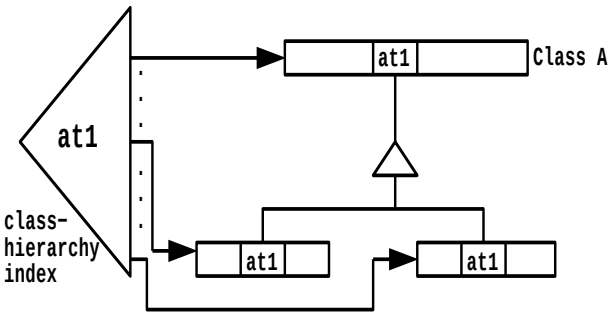



Figure 3.4: Class-hierarchy index on class A.

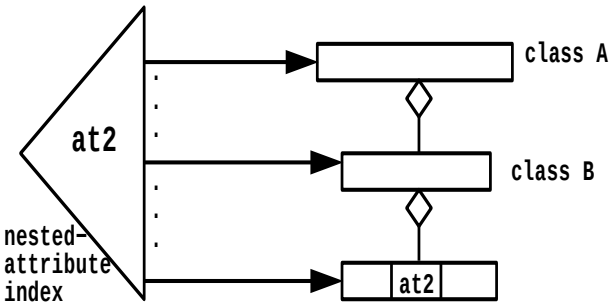



Figure 3.5: Nested-attribute index on attribute at2.

studied performance levels for each type, and for queries as well as updates. A particular result of this performance comparison is that nested-attribute (nested index) is preferred over class-hierarchy (path index) or two-dimensional index (multi-index) for the majority of queries.

It is clear that the indexing problem in OODBMS is to maintain primary indexes over the primary key of an object which is its *ObjId*. In our case, we need to consider secondary keys defined by the spatial location of a spatial object, and/or by date in temporal objects. Not all objects in a OODB are spatial, temporal, and/or spatio-temporal, so a primary index is needed to locate, to retrieve, and to maintain objects in the system. Additionally, it is possible to include special indexes

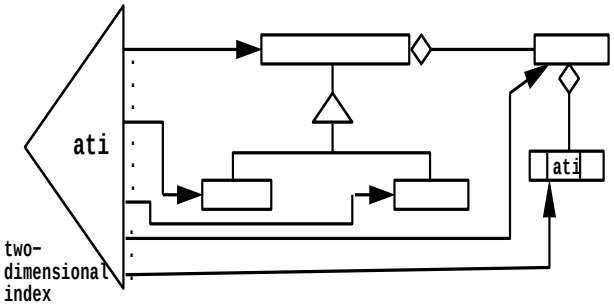



Figure 3.6: Two dimensional index.

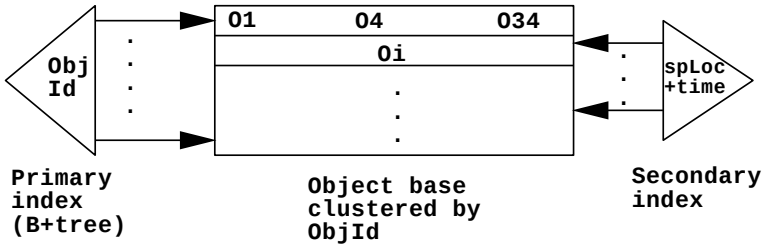



Figure 3.7: Primary and secondary indexes in an object base.

to support spatio-temporal access. If the spatial key and/or the temporal one are considered secondary in a specific application, then objects cannot be clustered by these keys because the main index clusters objects by class. That situation is shown in figure 3.7. Conversely, if the spatio-temporal key is the primary key in the application, spatio-temporal objects are then clustered by their location in space and associated date. Clustering is a DB technique used to store a group of objects physical close together on disk permitting a fast and an efficient retrieval operation.

In the case of the D/K model, the object identifier index is supported by using a prefix B+_tree [Ram94], which is a special case of the B_tree that improves it for cases having large prefixes within the keys. A prefix is a string forming the first part of a compound word, i. e. D-OKIM and D-OLAN have as a prefix D-O. The D/K model presents large prefixes within its *ObjId* which is composed of the class number and the instance number of the objects within its class. Thus, all of the instances of a class are clustered on disk and the prefix stored on the index pages of the $B^+$_tree is the class number. Spatio-temporal keys will be treated in the next sections.

The 1D search operations use two functions called exact match and range. The *exact match* permits the retrieval of an object which completely matches the key of search. The *range search* permits the retrieval of all objects whose ObjId is in the range of search. Examples of these two searches are *Find the object associated with this ObjId* (exact match), and *Find the objects of class XX* (range). These search functions are used intensively by the query language of the OODBMS. A summary of some OODBMS is included in appendix A where some interesting parameters are defined and presented as tables.

## 3.2 Spatial objects

The term *spatial* refers to space or relations of objects in space including both two and three dimensions. According to D. J. Peuquet in [Peu85] 'the term spatial data applies to any data concerning phenomenon areally distributed in two-, three-, or N-dimensions'. These dimensions are orthogonal and homogeneous. Further, she points out, 'geographic data, more specifically, are spatial data which normally refer to data pertaining to the earth... the term "geographic" data may also apply to data pertaining to other planets and objects in space'. One of the main application that uses spatial objects is geographic information systems (GIS), according to S. A. Roberts et al. in [RGHH91], 'a geographic information system (GIS) is an information system that stores spatially referenced data and that makes provision for the spatial representation of data to be displayed graphically'. In GIS, space refers to geographic space where geographic objects are related.

Spatial data can be categorized into several types. First, *point data* is the category where each data element is associated with a single location in 2D or 3D space, such as locations of cities in a map at appropriate scales. Second, *line data* is associated with a string of spatial coordinates representing isolated lines, elements of tree structures, or elements of network structures. The third type is *polygon data* which is associated with areas over a defined space. Polygons can be subdivided into: isolated polygons (no point common to two or more polygons), adjacent polygons (at least one boundary segment shared by two polygons), and nested polygons (one or more polygons are placed inside other polygons). Fourth, *some mixture of the above types* includes different line structures mixed with points or mixed with polygons.

In general, spatial models are considered to be linear in each spatial dimension, and space is regarded as discrete or continuous. Space can be bounded or unbounded, but in the majority of GISs, a bound as well as a range is assumed. A bound forms the boundaries or limits of a space. A range varies between this bound. Normally, it is assumed a bounded space where a spatial range applies. Space can also be absolute or relative, e.g. 52.62$^{o}$ latitude, and 6.18$^{o}$ longitude as absolute coordinates, and -4$^{o}$ west as a relative coordinate. Space has a distance function, called here *location distance* (LD(x,y)), satisfying the following properties:

**i)** $LD(x,y) \geq 0 \ \forall \ x,y$

**ii)** $LD(x,x) = 0 \ \forall \ x$

**iii)** $LD(x,y) = LD(y,x) \ \forall \ x,y$

**iv)** $LD(x,y) \leq LD(x,z) + LD(z,y) \ \forall \ x,y,z$

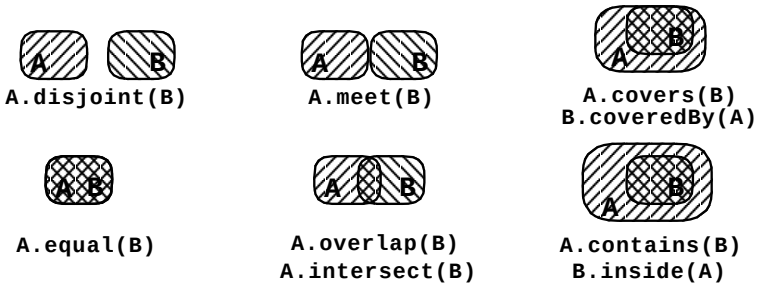



Figure 3.8: Topological relationships among MBRs.

As pointed out by R. Snodgrass in [Sno92] "Information that is spatially indeterminate can be characterized as *don't know exactly where* information". Spatial information can be indeterminate depending on system granularity. The spatial system granularity can be set to one metre and all of the spatial location identifiers have to be expressed in this unit. This system has both a physical and a logical representation. The physical representation is expressed via a set of bits and the logical one is the meaning of each bit pattern. In the majority of the systems, the metre is the unit adopted, which is defined as *the length of the path traveled by light in vacuum during a time interval of 1/299,792,458 of a second* [Sno92]. A physical representation of 32 bits/dimension permits a granularity of one decimeter for two dimensions and moving up to 64 bits reduces the granularity to a nanometer. This physical representation is used to choose the type of the spatial location key for implementation purposes, and we will use 32 bits per dimension.

### 3.2.1 Spatial relationships

Spatial entities are related by many relationships, such as: topological (disjointness, containment, overlapping, coverage), directional (above or north, below or south, left or east, right or west), and proximity (near, far, between).

Topological relationships have been treated in many references by M. Egenhofer [EF88], [Ege89], [EH90], [EF91]. The topological interactions are described by Egenhofer and Herring in [EH90] and reproduced here in table 3.1, where the object-oriented notation is used to call the operator or function, that represents the associated relationship, i.e. if *k.disjoint(m)=true*, then objects $k$ and $m$ are disjoint. The symbol $b$ means the border of the spatial object and $i$ means its interior. Each of these describes the intersection result cataloged as empty ($\emptyset$) or non-empty ($\neg\emptyset$). Intersections are done between boundaries ($b \cap b$), interiors ($i \cap i$), and its combinations. Figure 3.8 shows the graphic representation of these relationships.

| *Relationships* | b∩b | i∩i | b∩i | i∩b |
|---|---|---|---|---|
| k.disjoint(m) | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| k.contain(m) | $\emptyset$ | $\neg\emptyset$ | $\emptyset$ | $\neg\emptyset$ |
| k.inside(m) | $\emptyset$ | $\neg\emptyset$ | $\neg\emptyset$ | $\emptyset$ |
| k.meet(m) | $\neg\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| k.equal(m) | $\neg\emptyset$ | $\neg\emptyset$ | $\emptyset$ | $\emptyset$ |
| k.cover(m) | $\neg\emptyset$ | $\neg\emptyset$ | $\emptyset$ | $\neg\emptyset$ |
| k.coveredBy(m) | $\neg\emptyset$ | $\neg\emptyset$ | $\neg\emptyset$ | $\emptyset$ |
| k.overlap(m) | $\neg\emptyset$ | $\neg\emptyset$ | $\neg\emptyset$ | $\neg\emptyset$ |

Table 3.1: Topological relationships among intervals (1D) and MBRs (2D).

Directional or positional relationships are mentioned by W. Kim, J. Garza, and A. Keskin in [KGK93]. This set of relationships is presented in table 3.2, where [xi,xs,yi,ys] represents the coordinates defining an mbr, and each part is referred to as *object.part*. An object $k$ is above another object $m$ if *k's* mbr intersects the big mbr=[$-\infty$, $\infty$, m.ys, $\infty$]. Similar assumptions are made for left, right, and below. An object $k$ is directly above another object $m$ if *k's* mbr intersects the mbr=[m.xi, m.xs, m.ys, $\infty$]. Directly on the left, on the right, and below are calculated in a similar manner. An object $k$ is above and on the left of another object $m$ if *k's* mbr intersects the mbr=[m.xs, $\infty$, m.ys, $\infty$]. The rest of the relationships are determined in a related way. If we consider a bounded space, then $\infty$ is changed by the maximum value and $-\infty$ by the minimum value in this dimension. Figure 3.9 presents the graphic representation of each directional relationship.

| *Relationships* | *Calculation* |
|---|---|
| k.above(m) | $k.yi \geq m.ys$ |
| k.left(m) | $k.xi \geq m.xs$ |
| k.below(m) | $k.ys \leq m.yi$ |
| k.right(m) | $k.xs \leq m.xi$ |
| k.directAbove(m) | k.above(m)∧m.x.cover(k.x) |
| k.directLeft(m) | k.left(m)∧m.y.cover(k.y) |
| k.directBelow(m) | k.below(m)∧m.x.cover(k.x) |
| k.directRight(m) | k.right(m)∧m.y.cover(k.y) |
| k.aboveLeft(m) | k.above(m)∧k.left(m) |
| k.belowLeft(m) | k.below(m)∧k.left(m) |
| k.belowRight(m) | k.below(m)∧k.right(m) |
| k.aboveRight(m) | k.above(m)∧k.right(m) |

Table 3.2: Directional relationships among MBRs (2D).

Proximity relationships are also mentioned in [KGK93], and we only choose

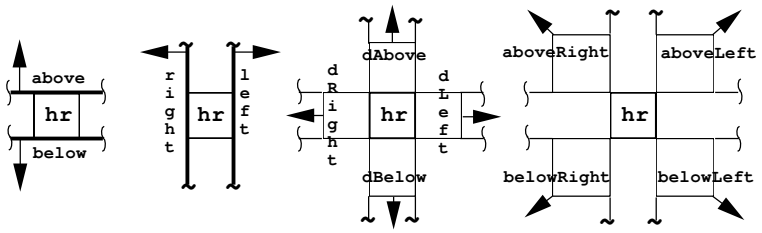



Figure 3.9: Directional relationships among MBRs.

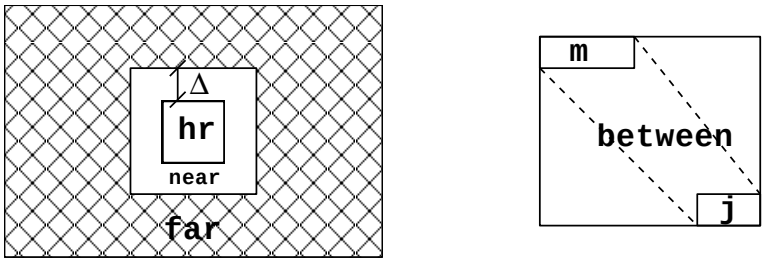



Figure 3.10: Proximity relationships among MBRs.

three of them, namely *near, far,* and *between.* The difficulty with these types of spatial relationship is the definition of cutoff values that can be used for specifying what is considered to be near and what is considered to be far. These relationships are presented in table 3.3, where we interpret proximity relationships in terms of equivalent topological relationships and where $\Delta$ is the specified unit to calculate whether the mbr overlaps objects near to the search object. As a first approximation, we calculate the new mbr $p$ for the *between* operator as the mbr of $m$ *and* $j$, but this mbr is too big and the definition will be re-examined in the next chapter. In figure 3.10, these relationships are illustrated graphically.

| *Relationships* | *Calculation* | *p* |
|---|---|---|
| k.near(m) | k.overlap(p) | (Xim-Δ,Xsm+Δ,Yim-Δ,Ysm+Δ) |
| k.far(m) | k.disjoint(p) | (Xim-Δ,Xsm+Δ,Yim-Δ,Ysm+Δ) |
| k.between(m,j) | k.overlap(p) | (Min(Xim,Xij), Max(Xsm,Xsj), Min(Yim,Yij), Max(Ysm,Ysj)) |

Table 3.3: Proximity relationships among MBRs (2D).

### 3.2.2 Spatial databases

A spatial database is the term used to distinguish databases that contain spatial data and retrieval operations. A spatial system uses a spatial model to treat spatial objects. The most expanded application using spatial data is a GIS, and we choose it for developing this subsection.

In GIS, there are two types of representations for spatial data: vector and raster. The main characteristic of the spatial entity is its location in space, and the relationships between two spatial entities are generally very numerous and depend on our perception of reality. Thus, the definitions of these entities and their relationships tend to be inexact and context dependent. Nevertheless, there are several models for representing geographic data as pointed out by R. Laurini and D. Thompson in [LT92]. First, the traditional method, a map, which provides a convenient method of spatial data storage and management. Second, two models for storing image data in digital form; vector and tessellation models. Finally, the hybrid type which contains characteristics of both vector and tessellation data models. Between vector data models we can find several other models as follows: spaghetti model (a direct translation of the paper map), topological model which retains some spatial relationship), GBF/DIME (geographic base file/dual independent map encoding), and POLYVRT (polygon converter). Tessellation models are: grid and other regular tessellations (square, triangular and hexagonal meshes), nested tessellations (recursive tessellation of the plane, the main example being the quadtree), irregular tessellations (size, shape, and orientation of the cells is a reflection of those of the data elements), scan-line models (raster) is a special case of the square mesh and this is a format commonly used by mass digitizing devices, and peano-scan which is a family of curves that permit the transformation from N-dimensional space to line and vice versa. It is well known that both models can be used but we restrict our work to the vector model based on POLYVRT.

We are interested in vector object-oriented models. One of these types is the D/K model wherein is defined a group of classes that support spatial objects and spatial relationships. Figure 3.11 shows the D/K spatial classes and figure 3.12 presents the implementation of the basic, graph, and spatial classes of the D/K model already defined by Montilva. We need not extend these classes because this model contains all of the classes needed for supporting spatial objects. An application of this model is presented by J. Montilva and I. Besembel in [MB96].

Similar to the OODBMS, in spatial databases there are some extended versions of SQL to support spatial queries. One version is referred by W. Kim in [Kim95]. This extension emphasizes the use of two operations, the window operation and the spatial join. A *window operation* combines a spatial selection with a nonspatial selection. A *spatial join* involves more than one spatial attribute.

The data manipulation classifies a predicate as spatial if the condition involves at least one of the spatial attributes or a spatial operation. Spatial functions apply to

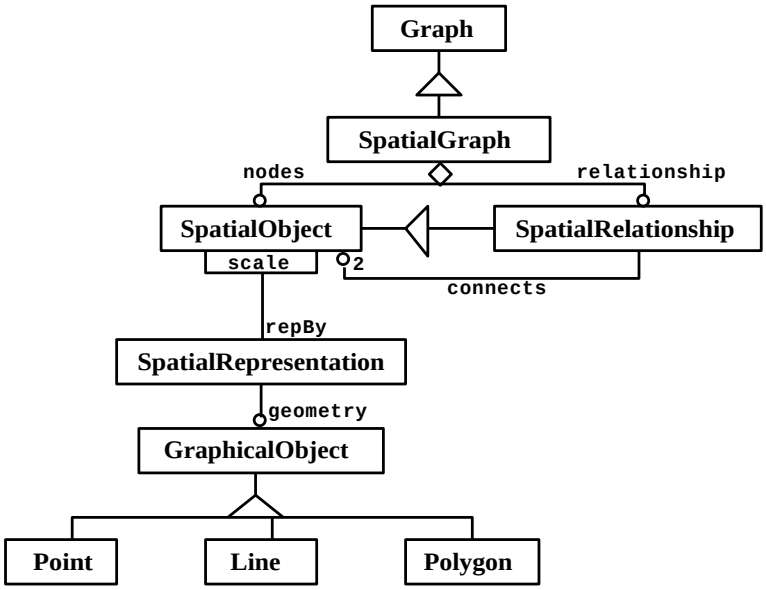



Figure 3.11: D/K spatial classes.

| | **Spatial Object** | **Spatial Graph** |
|---|---|---|
| **Spatial level** | SpObject<br>SpRepresentation<br>GraphicalObject<br>SpPolygon<br>SpChain<br>SpNode | SpDigraph / SpUndirGraph<br>SpArc / SpEdge<br>SpPath / SpChain<br>SpSimplePath / SpSimpleChain |
| **Graph level** | **Digraph** | **Undirected Graph** |
| | Digraph<br>Arcs<br>Path<br>SimplePath | UndirGraph<br>Edge<br>Chain<br>SimpleChain |
| **Basic level** | List, Set, IndexArc, Relationship | |

Figure 3.12: D/K classes implemented in a C++ library under UNIX platform.

spatial attributes such as: area, perimeter, centroid, object_at, in_circle, nearest_to, length, adjacent_to, intersect, in_window. An example of a spatial query is *Find all the objects in_window.*

There are some problems to extend the SQL to support spatial queries. As pointed out by Kim, "The problem is that SQL is primarily a means to retrieve from a tabular representation, while spatial applications often require retrieval from a graphical representation". Because of this difficulty, there is not a standard spatial extension to SQL equivalent to OSQL for object databases.

### 3.2.3 Indexing spatial objects

The main requirements for spatial access methods presented by A. Frank in [Fra91] are the following:

- The topological relations do not form a hierarchical structure. In our particular case, we will support topological relationships throughout using topological operators, we do not include these relationships in the access structure.
- It must be based on a general spatial object. We use the SpatialObject class of the D/K model, which is general.
- It must avoid the arbitrary division of any object. We shall not divide any spatial object, but we will permit overlapping at the level of the index nodes.
- Spatial objects are not spatially decomposable, and can have multiple representations. The SpatialObject class aforementioned supports multiple representations for a given spatial object and it does not decompose it.
- Versions and historic data are necessary in some GIS applications. This point will be treated in the next section.

In chapter 2, we studied several multi-dimensional access methods that can be used for indexing spatial objects. Following the requirements mentioned above and the comparative table of chapter 2, we choose the R_tree structure that will be implemented based on the object-oriented approach. Taking advantage of this approach, we will extend the spatial structure for supporting temporal access too, as we mentioned in the final section of chapter 2. In summary, the object-oriented R_tree will contain several spatial search functions, such as: topological, directional, and proximity. We will study in more detail these functions or operators in the next chapter for implementation and optimization purposes.

## 3.3 Temporal objects

Time is associated with objects in the majority of Information Systems. Time models can be linear or branching. In the linear model, time passes from the past to the future in an ordered manner as following a time line. Branching models consider a time line from the past to the present, and several time lines from the present to the future. The density of this line defines two types of model, discrete or continuous. A *discrete linear model* considers the time line to be isomorphic to natural numbers where each natural number is a unit of time (the smallest duration of time that can be represented in this model). In *continuous linear models*, the time line is isomorphic to real numbers where each real number corresponds to a *point* in time.

Time can be bounded having a beginning and an end. As a metric, time has a distance function that has the same four properties mentioned before for the space domain, and it is called *temporal distance* (TD). Time may be absolute or relative, e.g. 10:55 a.m. March 5th, 1996 or 11:50 a.m., respectively. Generally, time is considered to have two dimensions, one for valid time and other for transaction time. *Valid time* records the time a fact was true in reality, and *transaction time* records the time the fact was first recorded in the system. The *user-defined time* is also defined, which indicates that its values are only known by the user and not considered by the system. The two first dimensions are not homogeneous and they are considered to be orthogonal.

As mentioned by R. Snodgrass in [Sno92] "Information that is *historically indeterminate* can be characterized as *don't know exactly when* information". This indeterminacy applies only to valid time because transaction time is always determinate. The granularity of the transaction time line is the smallest inter-transaction time. For valid time, its time line granularity is set by the system. In the majority of systems, the chosen unit is the second, which has many definitions as mentioned in [Sno92]. The physical representation depends on the resolution needed and this imposes a granularity and a range, e.g. 8 bytes permits both a resolution of seconds and a range of 36 billion years, or a resolution of microseconds and a range of 17,400 years. Temporal systems need to support several calendars and transformation functions between them. We will use a physical representation of 8 bytes for temporal data.

A temporal object is an object-oriented object that has associated at least one temporal property which is represented as temporal data in one or more of the tem-

poral dimensions described above. For this kind of object, we study the relationships between them in the next section.

### 3.3.1 Temporal relationships

The first reference to the use of some special temporal comparison operators is by J. Allen in [All83], Navathe and Ahmed in [NA88] refer to these operators to be used in a language interface that they propose. These operators, combined with those presented by R. Barrera and K. Al-Taha in [BAT90], are shown in table 3.4. As explained in previous tables, $b$ indicates the border of the interval, and $i$ represents the interior. For the *adjacent, follows, and precedes* functions, it is necessary to differentiate between inferior (bi), and superior border (bs). These temporal relationships permit the representation of any relationship that may hold between two intervals and they can be used as temporal operators or functions in temporal queries. Figure 3.13 shows these relationships in a graphic manner.

| *Relationships* | b∩b | i∩i | b∩i | i∩b | $b_i \cdots b_s$ |
|---|---|---|---|---|---|
| q.before(t) | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | q.$b_s$ <t.$b_i$ |
| q.after(t) | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | q.$b_i$ >t.$b_s$ |
| q.during(t) | $\emptyset$ | $\neg\emptyset$ | $\neg\emptyset$ | $\emptyset$ | |
| q.overlap(t) | $\emptyset$ | $\neg\emptyset$ | $\neg\emptyset$ | $\neg\emptyset$ | |
| q.tMeet(t) | $\neg\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | |
| q.equal(t) | $\neg\emptyset$ | $\neg\emptyset$ | $\emptyset$ | $\emptyset$ | |
| q.start(t) | $\neg\emptyset$ | $\neg\emptyset$ | $\emptyset$ | $\neg\emptyset$ | |
| q.finish(t) | $\neg\emptyset$ | $\neg\emptyset$ | $\neg\emptyset$ | $\emptyset$ | |
| q.adjacent(t) | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $t.b_i - q.b_s$ = TU $\cup$ $q.b_i - t.b_s$ = TU |
| q.follow(t) | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $q.b_i - t.b_s$ = TU |
| q.precede(t) | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $t.b_i - q.b_s$ = TU |

Table 3.4: Temporal relationships among intervals (1D). (TU: time unit)

The implementation of these relationships as functions is more straightforward than for the corresponding spatial relationships, being one rather than two-dimensional. We take advantage of this by developing a temporal indexing extension of the R_tree as a base for building further dimensions.

### 3.3.2 Temporal databases

A temporal database models the dynamically changing world, tracing events and its related data [Lan89]. An atemporal database is a snapshot of the lastest available

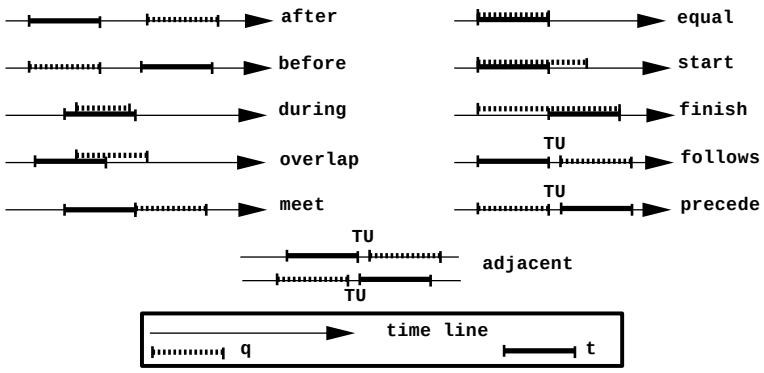



Figure 3.13: Temporal relationships among intervals.

data. Temporal databases have many domains of applications, such as: business planning, control processes, maintaining medical, sports and legal case histories, business activities, scientific experiments, policy analysis, census, and so on. Early work in this area mentioned some ideal requirements concerning historical objects, events, time-stamps, and the use of more than one temporal dimension [Lan89], [Ari87], [CR87].

The two dimensions, valid time and transaction time, can be supported by temporal database management systems (TDBMS). As defined by Snodgrass in [Kim95], there are four database models:

1. A snapshot database model supports neither valid or transaction time.
2. A valid-time database model supports only valid time.
3. A transaction-time model supports only transaction time.
4. A bitemporal database model supports both valid and transaction time.

Temporal databases correspond to this last model because they need both valid and transaction time. For our case, we first consider the historical database model by providing a dimension of the multi-dimensional access method to contain valid time values. In a second consideration, we include another dimension in the structure to support transaction time. The inclusion of user-defined time is considered and supported by most of the commercial DBMSs by providing a date-time domain that can be associated with object attributes, which can be indexed explicitly by the user.

The temporal data models can be categorised as event-oriented or time-based. The *event-oriented approach* uses the concept of event as its fundamental information from which any state of the database can be derived. It maintains an event database that can be queried for any particular time. The answer is generated by using specific

derivation rules which can obtain historical database states. One implementation of this approach is presented by S. Soukeras and P. King in [SK94]. They consider this approach better than the state-oriented one, due to the flexibility to expand the view it provides at any time, by deriving states from events raising the semantic level of the database and making it more straightforward to manipulate.

The *time-based approach* considers time as a separate dimension, similar to one-dimensional space. The different models of this approach can be either point-based, interval-based, or a mixed approach, as presented by R. Barrera and K. Al-Taha in [BAT90]. *Point-based models* consider a dense, complete, unbounded, and real time line. Points are used as its basic temporal objects. Each time interval consists of an ordered pair of points. *Interval-based models* consider a linear and discrete time line, and intervals are its primitives. It does not allow time points and it applies a temporal logic based on temporal intervals and first-order predicate calculus. Temporal logic operators allow specification of the time of validity of a fact either in the past or in the future [Val96].

The third type of temporal model, which is the *mixed model*, allows both time points and intervals by considering time points as zero length intervals. This model allows assertions over time intervals and it considers a linear and totally ordered time. We choose this model to be used in our index method because we already have the Interval class that can support time intervals.

According to Snodgrass in [Kim95], there are three approaches to incorporate time-varying objects in an OODBMS. The first approach is to use the OO model directly, and the user is charged with the task of managing the semantics of valid time, which is normally thought of as linear. The main advantage of this approach is that the user can specify the needed semantic of this time. The disadvantage is that the specification of the schema and queries have no special tools to deal with temporal information. The second approach is to tailor general features of the object-oriented data model to support time. Finally, the third approach incorporates time into the data model by providing special constructs to deal with time-varying objects. This last approach is mostly used in supporting histories through valid time, and versions through transaction time, as mentioned by L. Valet in [Val96].

In order to support temporal characteristics in the D/K model, we choose the third approach and present here an extension based on the definition of specific temporal objects, as illustrated in figure 3.14. The *TemporalObject* class provides the main facilities to define temporal characteristics to object structure and behaviour. The *TemporalRepresentation* class permits the use of several forms of the temporal

| Temporal | Temporal Graph |
|---|---|
| TemporalObject | TDigraph / TUndirGraph |
| TemporalRepresentation | TArc / TEdge |
| | TPath / TChain |

Figure 3.14: Extension to the D/K classes to support temporal objects.

data, corresponding to calendars and transformation functions. Finally, the *TDigraph* class supports the concept of time map proposed by Dean and McDermott in 1987 and mentioned by R. Barrera and K. Al-Taha in [BAT90]. A time map "is a graph whose nodes refer to points or instances of time corresponding to the beginning and ending of events". Arcs are labeled with a lower and upper bound of time, or time interval. A relation between any two points on the time map is represented by a path between them. We do not go deeper into the definitions of these classes because it is outside the objectives of this work.

Actual TDBMSs use extensions of two of the main query languages, namely SQL and QUEL. The temporal SQL (TSQL) presented by S. Navathe and R. Ahmed in [NA88], is the first extension of the SQL language to permit temporal analysis and definition in temporal databases. A version called TOSQL is defined as the DML for the temporal oriented data model presented by G. Ariav in [Ari86]. It is based on the notion of atomicity of events and it supports object and time selection.

Over a dozen temporal object-oriented query languages have been proposed in the bibliography as pointed out by Snodgrass in [Kim95]. The majority of them are based in relational query languages such as SQL. In particular, the TOSQL and TOOSQL are two representatives of SQL extensions that support temporal analysis. In TOOSQL, the **when** clause is added to the select instruction, which permits the use of temporal references in the query. An example of that type of queries is *Find all of the objects inserted in the database after dd-mm-yy* (where 'after dd-mm-yy' is the 'when' clause).

### 3.3.3 Object versioning

In OODB, there are two approaches for versioning, object versions and database versions. In the object versions approach, versions may be defined either for an object or for an existing version of an object. According to A. Björnerstedt and C. Hultén in [KL92], "an object version represents an identifiable state of an object. Object versions are either totally ordered as a function of time, or partially ordered in terms of a successor function". Version control defines two types of relationships,

the derived-from and the version-of. As pointed out by W. Kim in [Kim90], the *derived-from* relationship is between a new version and an old version of the object from which the new version was derived. The *version-of* relationship is between each version of an object and an abstract object that represent the object. Versions of an object form a digraph which represents the history of evolution of a versioned object. Any number of new versions may be derived from any version at any time. Generally, each version contains attributes that identify its version identifier, number, type, time of creation, time of last modification, and a list of references to the version identifiers of all versions directly derived from it. Additionally, users decide when the new version of the object is needed and she/he decides if it is a new version or a new object. Surrogates are used to identify each object version. In this approach, some researchers consider two levels of version management: application-level, and system-level. These levels are associated with the two time semantics called valid time and transaction time, respectively.

In supporting transaction time, two models of versioning can be used. According to Snodgrass, the *extension versioning* indicates versioned objects or their versioned attributes, and *schema versioning* provides versioned definitions of the objects. In this latter model, several schemes are stored in the database at different transaction times, and it can be viewed as the evolution of the database schema, where two types of management are possible. One of them permits all objects to be viewed in all of the schemes by performing the appropriate transformation. The other one associates objects with schemes allowing only objects defined in a particular schema to be present in this schema.

In the database version approach, objects are not isolated in the database, and a change of an object modifies the state of database. Additionally, the database integrity must be maintained along the sequences of changes of states. A version of an object is considered a modified state of the database and consequently, this modified state is a version of the entire database, that is a database version. A version of a database is created by derivation. To avoid object duplication a system of stamps is used to recognise object versions in each database version.

In summary, the support for versions within databases immediately takes us on to consider temporality. A database that supports time is more general that one which supports versions, so we will develop the temporal management and use this later for supporting versions.

### 3.3.4 Indexing temporal objects

In this context, temporal indexes are involved. As pointed out by Snodgrass, the majority of temporal index methods are based on $B^+$_tree, and he reports only two cases where R_trees are used. In these cases, the R_tree supports the two time dimensions, valid and transaction, and the temporal key is expressed as intervals.

In our case, the access method will be used to index temporal objects by following the mixed approach. It is the more appropriate because the hyper-rectangles used for the index entries are defined as intervals. We also consider the point-based approach by considering a point as an interval of zero length. As an initial step, we only considered one time dimension corresponding to valid time. In a second step, we include another dimension for transaction time. To use both dimensions, the user or the query processor must use the same operator but including a parameter to indicate which dimension is queried. More details will be shown in the next chapter.

## 3.4 Spatio-temporal objects

Basically, we define a spatio-temporal object as an object-oriented object that has at least one spatial and one temporal property. These properties are represented by spatial and temporal data, respectively. It can be represented by a four-tuple (objId, mbr, t, att) where *objId* is the object-oriented identifier assigned by the OODBMS to that object, *mbr* is the bounding rectangle that indicates the spatial location and shape of that object, $t$ is the description of the temporal characteristics of that object, and *att* is a set of other attributes that describes the object.

Recalling, an mbr in 2D is represented by four values corresponding to the lower left corner of the rectangle specified by the two first values and the upper right corner of the mbr formed by the two last values.

Following the paper of Seger and Shoshani in [SS88], the temporal characteristics identified by $t$ are composed of: *granularity* which says if the representation is ordinal or calendar; *life-span* that contains two values represented as ordinal or calendar for the start point and the end point; *regularity* that indicates a regular time sequence or an irregular one; and *type* which can be constant, continuous, discrete or user defined. However, only the life-span of $t$ is considered, in our spatio-temporal index because it is the specific temporal value that we need to express time. A time sequence is defined by an ordered sequence formed by the temporal data values $t$ corresponding

to a spatio-temporal object. Similarly to the time sequence, a spatial sequence can be defined which is formed by the spatial component of the mbr corresponding to a spatio-temporal object. The relationship between time and spatial sequences will be defined more carefully in chapter 4.

As mentioned by R. Snodgrass in [Sno92], time and spatial models can be discrete, continuous or dense, and "in all three of these alternatives, two separate space-filling objects cannot be located in the same point in space and time".

A distance function where both spatial-distance and time-distance are involved can be defined. For example, a spatio-temporal distance (LTD) can be defined to be used in *near-far* spatio-temporal queries. With the distance functions, both coincidence and neighbouring functions can be specified including a sufficiently close parameter which depends on the application context. The main disadvantage of this view is the homogeneous treatment of both spatial and time dimensions.

### 3.4.1 Spatio-temporal relationships

We consider three groups of relationships, namely: spatio-temporal, spatial, and temporal. In the spatio-temporal, we only consider the topological relationships among d-dimensional rectangles, called here hyper-rectangles, where spatial and temporal dimensions are treated homogeneously. The $d$ dimensions defined for spatio-temporal objects are internally represented by using a unique format unifying data representation for homogeneous treatment. Table 3.5 presents the same relationships that have been shown in table 3.1, but here topological relationships are considered for dD mbrs (hyper-rectangles).

| *Relationships* | b∩b | i∩i | b∩i | i∩b |
|---|---|---|---|---|
| k.disjoint(m) | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| k.contain(m) | $\emptyset$ | $\neg\emptyset$ | $\emptyset$ | $\neg\emptyset$ |
| k.inside(m) | $\emptyset$ | $\neg\emptyset$ | $\neg\emptyset$ | $\emptyset$ |
| k.meet(m) | $\neg\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| k.equal(m) | $\neg\emptyset$ | $\neg\emptyset$ | $\emptyset$ | $\emptyset$ |
| k.cover(m) | $\neg\emptyset$ | $\neg\emptyset$ | $\emptyset$ | $\neg\emptyset$ |
| k.coveredBy(m) | $\neg\emptyset$ | $\neg\emptyset$ | $\neg\emptyset$ | $\emptyset$ |
| k.overlap(m) | $\neg\emptyset$ | $\neg\emptyset$ | $\neg\emptyset$ | $\neg\emptyset$ |

Table 3.5: Topological relationships among minimum bounding hyper-rectangles (MBHR).

Spatial and temporal relationships were already presented in the aforementioned sections. Spatial operators treat only spatial data and temporal operators are only

| Spatio-temporal Objects | Spatio-temporal Graph |
|---|---|
| STObject | STDigraph / STUndirGraph |
| | STArc / STEdge |
| | STPath / STChain |

Figure 3.15: Extension to the D/K classes to support spatio-temporal objects.

applied on temporal data, respectively. It is possible to combine spatial and temporal relationships to be used to answer spatio-temporal queries. These possibilities will be treated in the next section.

### 3.4.2 Extension proposition for the D/K model to support spatio-temporal objects

We follow the third approach mentioned by Snodgrass in [KGK93], to incorporate time into the D/K model. We propose an extension that supports special constructs to deal with time-varying objects. Figure 3.15 shows two new groups of object classes or types, such as: spatio-temporal objects and graph. An object instance of the STObject inherits spatial and temporal representation, but internally is stored in a uniform manner by applying transformation functions to it. Thus, a STObject has a unique representation, which unifies its representation. Figure 3.16 presents the summary of the D/K model extension proposal by using an OMT diagram. It indicates that a spatio-temporal graph is a graph composed of nodes which are spatio-temporal objects and arcs (edges) that are spatio-temporal relationships other than topological ones.

The internal specification of these new classes in the D/K model are outside this work. We only include this example of extension here for showing how the D/K model should support spatio-temporal objects.

### 3.4.3 Querying and indexing spatio-temporal objects

The object-oriented spatio-temporal query language (OOSTSQL) is an extension of SQL proposed by Cheng and Gadia, and mentioned by Snodgrass in [Kim95]. There is no other reference to a query language supporting spatio-temporal queries. In this language, the select clause mentioned in sections 3.2.2 and 3.3.2 deals with spatial and temporal attributes. These temporal data can be either valid or transaction time.

Query processing of spatio-temporal objects consists of spatial query conditions

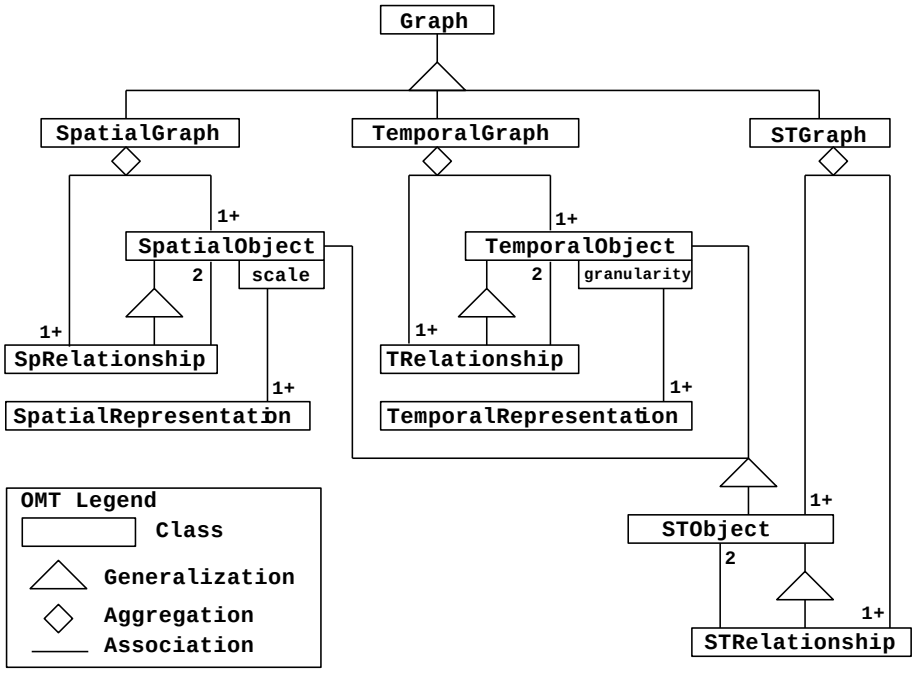



Figure 3.16: Extension proposition of the D/K model to support temporal and spatio-temporal classes.

and temporal query conditions, e.g. Which object overlaps a query region from time t1 to t2?. Because the R_tree is an indexing method that can only retrieve object identifiers, it is not possible to query it directly by the user. Thus, the operators included in the R_tree class will only be used by the system through the query optimizer and processor, which has to decompose a user query into a query plan to be executed by the processor.

## 3.5 Concluding remarks

The definition of spatio-temporal objects takes us to a deeper study of the three kind of objects, namely: object oriented, spatial, and temporal. By understanding their characteristics and relationships, we realise that each kind can be treated alone or all together. For each kind of object, we defined operators that support its relationships, and we realised the multiple possibilities and advantages that the inclusion of these operators gives to the R_tree class. However, it seems to us that there are too many operators for that inclusion, so we will present in the next chapter, a deeper analysis to try to get a minimal set of these operators.

The objective of the inclusion of the spatio-temporal operators is to improve the performances of the index method by increasing the hit ratio of the retrieval

operations. In this context, the proposed *RTree* class will permit the main retrievals needed by the D/K query language, and will support spatial (topological, positional, and proximity), temporal, and spatio-temporal operators in the context of MBRs. The query processor of the DML may use these operators to retrieve a set of objects that contains the answer of a specific query. Since the R_tree structure is built with mbhr, the set of retrieved objects should be greater than the set of answering objects for a given query. Special calculations are always needed, called computational geometry, to exactly determine the set of answering objects from the given set of retrieved objects.

Special attention was paid to the versioning of objects in OODB, because in both of the mentioned approaches, time is chosen to order object versions. OODBMS can support versioning without temporal analysis by storing the creation time of the version. Additionally, users decide when the new version of the object is needed and she/he decides if it is a new version or it is a new object. Surrogates are used to identify each object version and this surrogate does not take into account its creation time. The support of creation time means that the system also supports transaction time. The R_tree index cannot support versioning without the indexing of transaction time, and even with that, the definition of the D/K model of versioning is needed. This model can be extension versioning or schema versioning. If the extension versioning model is chosen, then the schema versioning may or may not be supported. We think the support of both models of versioning is the best alternative for the D/K model.

The R_tree class will support the two time dimensions that can be used in the versioning model. We do not go further into this point because we consider it is outside of the objective of this work. We hope that the D/K query language wisely uses the available operators to improve data retrieval times. The next chapter will present the minimal set of the operators to be included.

# Chapter 4

# R_tree formal model

Retrieving multidimensional data (spatial or multipoint) for supporting non-standard database applications, such as Computer Aided Design (CAD)or Geographic Information Systems (GIS), among others, has been treated in many references as mentioned in chapter 2. The R_tree structure [Gut84] is a generalization of the B_tree [BM72] to higher dimensions. This chapter presents an object-oriented formal model of this structure. At the beginning, we include some design characteristics to better explain the bases of our formal model, and we finalize with the presentations of the requirements for the **RTree** class extensions.

This chapter is organised as follows: section 4.1 presents the main concepts of the R_tree structure as it was proposed by A. Guttman in [Gut84]. Section 4.2 shows the formal definitions to base the R_tree spatio-temporal extensions. An object calculus based on those presented by D. Straube and M. Tamer in [ST90] is shown in section 4.3. The illustration of the R_tree searches extended with the spatial, temporal, and spatio-temporal cases is treated in section 4.4. Based on the described formal model, section 4.5 presents the requirements of the **RTree** class that will be designed and implemented in the next chapter, and finally, section 4.6 contains some concluding remarks.

## 4.1 The R_tree structure

An R_tree is a height-balanced tree whose leaf nodes contain pointers to data objects. The index is fully dynamic, where insertions and deletions may be intermixed with searches without the need for periodic reorganization. According to A. Guttman in [Gut84], an R_tree satisfies the following properties:

**(1)** Every leaf node contains between $m$ and $M$ index records unless it is the root.

**(2)** For each index record **(I, tuple-identifier)** in a leaf node, **I** is the smallest rectangle that spatially contains the n-dimensional data object represented by the indicated tuple.

**(3)** Every non-leaf node has between $m$ and $M$ children unless it is the root.

**(4)** For each entry **(I, child-pointer)** in a non-leaf node, **I** is the smallest rectangle that spatially contains the rectangles in the child node.

**(5)** The root node has at least two children unless it is a leaf.

**(6)** All leaves appear on the same level.

In the above, $M$ is the maximum number of entries in one node and $m \leq \lceil M/2 \rceil$ specifies the minimum number of entries in a node. As in the B_tree, $m$ is named the R_tree order. Each R_tree node is equivalent to a disk page called here a *blob*. $M$ and $m$ depend on the *blob* size (blobSize). For a given blobSize, $M$ is equal to the blobSize divided by the length of the entries (entLength), i.e. $M = \lceil blobSize/entlength \rceil$. Figure 4.1 presents an R_tree with its overlapping regions.

Each leaf node contains a set of leaf entries, and branch nodes contain branch entries. Leaf and branch entries within nodes have the following format: **[hr, bn]** where **hr** is a $d$-dimensional bounding rectangle formed by $d$ intervals of the form **[inf, sup]** describing the minimum and maximum value in each dimension, and **bn** is the blob identifier that refers to either the blob which contains the tuple identified by **hr**, or the blob which contains a leaf or branch node of the R_tree. The minimum bounding hyper-rectangle (*mbhr*) that covers the object is represented by **hr**. An *mbhr* has its edges parallel to the axes of the data space. The difference between a leaf and a branch node is indicated by an attribute value stored in the head of a node that is called *level*, because it indicates the height of a node defined by the path length from this node to a leaf node. Each node has a head and the rest of the node. The head contains the six attribute values, current blob number indicating where the node is stored (blobId), height of the node (level), current number of entries (entries), current length of the node in bytes (length), blob number of the anterior

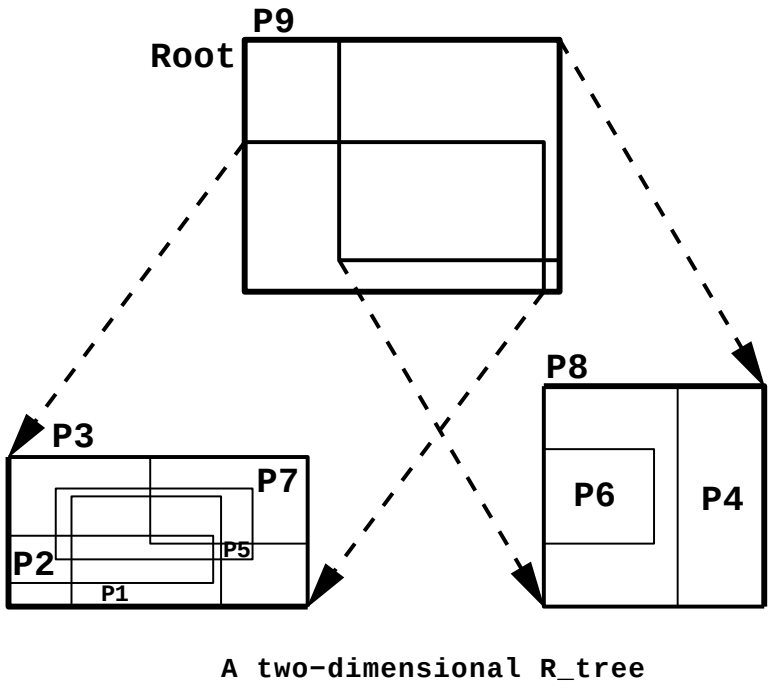



Figure 4.1: An R_tree structure.

node (left), and blob number of the posterior node (right). The rest of a node is composed of a set of entries. Figure 4.2 illustrates the node format corresponding to either leaf or branch nodes.

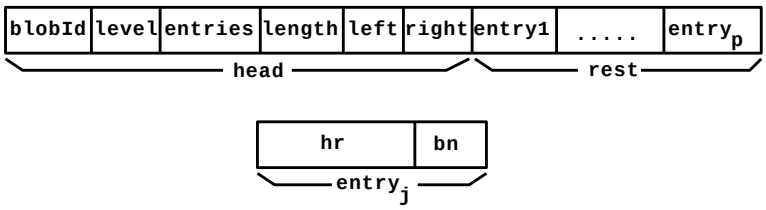



Figure 4.2: Leaf and branch nodes format.

Both a node of the tree and an object in the object base are stored using the same structure, that is the *blob*. We use this structure because the object base can contain many different types of objects, especially those corresponding to multimedia or image objects defined in the D/K model. The *blob* is managed in multimedia systems, and it normally has a fixed length between 2KB and 4KB. To manage and support *blobs*, the **Blob** class is used as an important class related to the **RTree** class. The format of a *blob* is almost the same as for R_tree nodes. Figure 4.3 shows the blob format used. A *blob* has a head and a tail. The head contains the following attributes: the status of the blob (status), says if the blob is actually in use or not,

the current number of objects (objNum), the current length in bytes (length), its left brother blob (left), and its right brother blob (right). The tail contains a finite set of objects. The internal behaviour of this class should be defined in the object manager system, so we only use this class assuming that each indexed object has a blob number associated, and this number is retrieved by using the R_tree index.

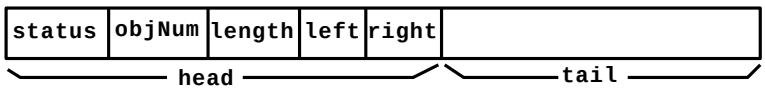



Figure 4.3: The blob format.

To insert a new entry in the R_tree, two steps have to be made. The first is a search from the R_tree root node to the corresponding leaf node storing the descendent route of nodes into a stack without the need of using a reference to the father node into each child node. The second step is insertion of the new entry into the node if there is enough room in it, otherwise the splitting function is invoked, that moves half of the existing entries into a new node according to the splitting policy adopted. The original paper proposes three splitting policies. One of them is that of the minimal covering area for the two groups of entries, those that remain in the split node (old node) and those that will be put into a new node. We choose to make the new node a right brother of the existing node. A new entry is generated to cover all of the entries of the new node, and this has to be inserted into the father node of the old node which must, in turn, be split if already full. This process can reach the root node provoking the growth of the R_tree, just as in the B_tree.

To delete an existing entry, it is also a two stage process. The first step is to find the entry in a leaf node. If this entry does not exist, the process is finished, otherwise the second step is an entry deletion. After a deletion, it is checked if the leaf node is underflowed or not. Underflows are treated by re-insertion of the entries remaining in the underflowed node until that node is empty at which point it can be deleted. The corresponding covering entry in its father node has to be deleted too by only following the second step. Obviously, this process can also reach the root node provoking the shrinking of the R_tree.

These two main operations or functions of the R_tree permit the dynamic maintenance of the structure. To retrieve objects through this access method, at least two other main functions are needed, the exact match search and the range search functions. The first finds the object reference of the target object that matches the multidimensional key given for searching. The second finds a set of object references

that corresponds to those target objects which are contained in the multidimensional search region.

In the following section, we present the R_tree extensions that expand the range search function for several more specific range search functions depending on topological, directional, or proximity searches.

## 4.2 Formal definitions

Spatio-temporal queries require the support of spatio-temporal relationships among spatio-temporal objects. These queries are treated by the query processor that analyzes, optimises, and processes them. Normally, a query is transformed into a query execution plan that contains a sequence of operations over a set of target objects. The main idea is to use the most convenient set of operators to restrain the search space of the target objects when the query processor executes a given query plan. Thus, it is important that the index structure supports a set of specific search operators in order to accelerate objects retrieval. With this aim, we define a formal model to clarify and state the set of operators that extend our R_tree funcionalities.

All definitions in our model are based on the model proposed by D. Straube and M. Tamer in [ST90], who define the following sets:

- a finite set of basic *domains* $D_1, \ldots, D_n$ where $D = \bigcup_{i=1}^{n} D_i$;
- a countably infinite set $A$ of symbols, called *attributes*;
- a countably infinite set *Oid* of object identifiers;
- a finite set *CN* defined by *CN* ={*Interval, Hyper-rectangle, SpatialObject, TemporalObject, and STObject*} class names;
- a finite set *MN* of method names.

Straube and Tamer define three types of values:

**Definition 4.1 (Values:)** *There are three types of values:*

1. *Every $v \in D$ is an atomic value for which there exists a textual representation*
2. *Every finite subset of Oid is a set value*
3. *Every element in $\mathcal{P}(\mathcal{A}) \times \mathcal{P}(\mathcal{D}) \times \mathcal{P}(Oid)$[1] is a structural value*

[1] $\mathcal{P}$ (X) denotes the powerset of X.

We restrict our initial work to atomic values.

The symbol $\vartheta$ denotes the set of all values, $\mathcal{I}$ denotes the set of all integer values, and $\Re$ is the set of all real values.

**Definition 4.2 (Objects:)** *An object is a triple o = (oid, cn, val) where oid* $\in$ *OId, cn* $\in$ *CN, and val* $\in \vartheta$. $O = OId \times CN \times \vartheta$ *is the set of all objects. The notation o.oid, o.cn, and o.val, denotes the object identifier, the class, and the value of object o, respectively.*

**Definition 4.3 (Interval:)** *An interval is an object where o.cn=Interval and it contains two atomic values* $\in \Re$ *denoted by inf and sup. It can be either open if it does not contain their borders, denoted by OpInt = (inf, sup); or closed if it contains their borders, denoted by CloInt = [inf, sup]. For both type of intervals, inf* $\leq$ *sup always. The notation o.inf and o.sup denote the inferior or minimum value and the superior or maximum value of the interval, respectively.* $Int \subset O$ *is the set of all intervals.*

**Definition 4.4 (Hyper-rectangle:)** *A hyper-rectangle is an object where o.cn = Hyper-rectangle and it contains an array of d intervals defining a multidimensional rectangular region in a multidimensional space (hr). The notation o(dim) denotes the closed interval in dimension dim that contains the extent of the hyper-rectangle in this dimension with* $0 \leq dim < d$. $HR \subset O$ *is the set of all hyper-rectangles.*

**Definition 4.5 (Spatial objects:)** *A spatial object is an object where o.cn = SpatialObject and it contains the number of spatial dimensions, denoted by o.sd, and the minimum bounding hyper-rectangle in the spatial dimensions that defines its spatial location (o.sl).* $SO \subset O$ *is the set of all spatial objects. The notation o.sl denotes the minimum bounding hyper-rectangle formed in the spatial dimensions of o, and o.ds denotes the spatial unit used in proximity spatial searches.*

**Definition 4.6 (Temporal object:)** *A temporal object is an object where o.cn = TemporalObject and it contains the number of temporal dimensions (o.td), and the minimum bounding hyper-rectangle in the temporal dimensions that defines its location in time (o.tl).* $TO \subset O$ *is the set of all temporal objects. The notation o.tl denotes the minimum bounding hyper-rectangle formed in the temporal dimensions of o, and o.dt denotes the time unit used in proximity temporal searches.*

**Definition 4.7 (Spatio-temporal object:)** *A spatio-temporal object is an object where o.cn=STObject and it contains the number of dimensions (o.d=sd+td), and*

*the minimum bounding hyper-rectangle in the spatio-temporal dimensions that defines its location in space and time (o.stl). $STO \subset O$ is the set of all spatio-temporal objects. The notation o.stl denotes the minimum bounding hyper-rectangle formed in the spatio-temporal dimensions of o.*

The symbols $\mathcal{ST}$ denotes $STO \cup SO \cup TO$, $SR$ denotes the set of all spatial relationships between spatial objects, $TR$ denotes the set of all temporal relationships between temporal objects, and $STR$ denotes the set of all spatio-temporal relationships between spatio-temporal objects.

We define the following function mappings to be used on search operations.

**Definition 4.8 (Boundary of an object:)** *Let $p$ be a dD point represented by $p = (p_1, p_2, \ldots, p_d), p \in \Re^d$, and an object $o_i \in \mathcal{ST}$. The dD point $p$ is in the boundary of $o_i$, $p \in b(o_i)$ if for some dimension $\delta$, $1 \leq \delta \leq d$, $p_\delta = o_i(\delta).inf$ or $p_\delta = o_i(\delta).sup$, and for all other dimensions $\varepsilon$, $1 \leq \varepsilon \leq d$, $\varepsilon \neq \delta$, $o_i(\varepsilon).inf \leq p_\varepsilon \leq o_i(\varepsilon).sup$.*

**Definition 4.9 (Interior of an object:)** *Let $p$ be a dD point represented by $p = (p_1, p_2, \ldots, p_d), p \in \Re^d$, and an object $o_i \in \mathcal{ST}$. The dD point $p$ is in the interior of $o_i$, $p \in i(o_i)$ if for all dimension $\delta$, $1 \leq \delta \leq d$, $o_i(\delta).inf < p_\delta < o_i(\delta).sup$.*

**Definition 4.10 (Spatio-temporal mbhr:)** *The minimum bounding hyper-rectangle of a spatial and temporal object is a mapping mbhr from the set of spatial and temporal objects STO to HR such that for all $o_i \in STO$, $mbhr(o_i)$ is the hyper-rectangle that completely covers the object $o_i$ in all d dimensions with $o_i.oid$.*

**Definition 4.11 (Spatial mbhr:)** *The minimum bounding hyper-rectangle composed of spatial dimensions is a mapping mbr from the set of spatio-temporal objects STO to HR such that for all $o_i \in STO$, $mbr(o_i)$ is the hyper-rectangle that completely covers the spatial dimensions of object $o_i$ in all sd dimensions with $o_i.oid$.*

**Definition 4.12 (Temporal mbhr:)** *The minimum bounding hyper-rectangle composed of temporal dimensions is a mapping t from the set of spatio-temporal objects STO to HR such that for all $o_i \in STO$, $t(o_i)$ is the hyper-rectangle that completely covers the temporal dimensions of object $o_i$ in all td dimensions with $o_i.oid$.*

For clarity, we use the following notations:

- The first and second dimensions representing spatial dimensions are denoted by $X$ and $Y$, respectively.

- The first and second dimensions representing temporal dimensions are denoted by $T$ and $TT$, respectively.

**Example 4.1**

- *o(X).inf denotes the minimum value of the first interval representing X coordinate corresponding to the first spatial dimension.*
- *o(Y).sup denotes the maximum value of the second interval representing Y coordinate corresponding to the second spatial dimension.*
- *o(T) denotes the first interval representing valid time corresponding to the first temporal dimension.*
- *If* $o_i \in SO$ *and* $o_i.sd = 2$, *then*
  $mbr(o_i) = ([o_i.sl(X).inf, o_i.sl(X).sup], [o_i.sl(Y).inf, o_i.sl(Y).sup])$ *with*
  $o_i.sl(X).inf \leq o_i.sl(X).sup$, *and* $o_i.sl(Y).inf \leq o_i.sl(Y).sup$.
- *If* $o_i \in TO$ *and* $o_i.td = 1$,
  $t(o_i) = ([o_i.tl(T).inf, o_i.tl(T).sup])$ *with* $o_i.tl(X).inf \leq o_i.tl(T).sup$.
- *If* $o_i \in STO$ *and* $o_i.sd = 2, o_i.td = 1, o_i.d = td + sd = 3$, *then*
  $mbhr(o_i) = (o_i.stl(T), o_i.stl(X), o_i.stl(Y))$,
  $mbr(o_i) = (o_i.stl(X), o_i.stl(Y))$, *and*
  $t(o_i) = o_i.stl(T)$.

The model presented by M. Egenhofer defines sixteen binary topological relationships, that are complete and are the result of the comparison of two objects in terms of their boundaries and interiors. The set of these relationships among intervals in one-dimensional space and among *mbhr*s in 2D was shown in the last chapter.

We can now establish the relationship between one object $o_i$ and another $o_j$, or one object $o_i$ and a rectangle $mbhr(o_i)$; or a rectangle $mbhr(o_i)$ and another $mbhr(o_j)$.

For each spatio-temporal relationship already shown in table 3.5, we define the following: Let $k$, $m$ be two spatio-temporal objects $\in STO$.

- *k.disjoint(m):* Returns true if k and m are disjoint, otherwise returns false. k and m are disjoint if
  $b(k) \cap b(m) = \emptyset,\ i(k) \cap i(m) = \emptyset,$
  $b(k) \cap i(m) = \emptyset,\ i(k) \cap b(m) = \emptyset$
- *k.contain(m):* Returns true if k contains m, otherwise returns false. k contains m if
  $b(k) \cap b(m) = \emptyset,\ i(k) \cap i(m) = \neg\emptyset,$
  $b(k) \cap i(m) = \emptyset,\ i(k) \cap b(m) = \neg\emptyset$
- *k.inside(m):* Returns true if k is inside m, otherwise returns false. k is inside m if
  $b(k) \cap b(m) = \emptyset,\ i(k) \cap i(m) = \neg\emptyset,$
  $b(k) \cap i(m) = \neg\emptyset,\ i(k) \cap b(m) = \emptyset$

- *k.meet(m):* Returns true if k meets m, otherwise returns false. k meets m if
  $b(k) \cap b(m) = \neg\emptyset,\ i(k) \cap i(m) = \emptyset,$
  $b(k) \cap i(m) = \emptyset,\ i(k) \cap b(m) = \emptyset$

- *k.equal(m):* Returns true if k is equal to m, otherwise returns false. k is equal to m if
  $b(k) \cap b(m) = \neg\emptyset,\ i(k) \cap i(m) = \neg\emptyset,$
  $b(k) \cap i(m) = \emptyset,\ i(k) \cap b(m) = \emptyset$

- *k.cover(m):* Returns true if k covers m, otherwise returns false. k covers m if
  $b(k) \cap b(m) = \neg\emptyset,\ i(k) \cap i(m) = \neg\emptyset,$
  $b(k) \cap i(m) = \emptyset,\ i(k) \cap b(m) = \neg\emptyset$

- *k.coveredBy(m):* Returns true if k is covered by m, otherwise returns false. k is covered by m if
  $b(k) \cap b(m) = \neg\emptyset,\ i(k) \cap i(m) = \neg\emptyset,$
  $b(k) \cap i(m) = \neg\emptyset,\ i(k) \cap b(m) = \emptyset$

- *k.overlap(m):* Returns true if k overlaps m, otherwise returns false. k overlaps m if
  $b(k) \cap b(m) = \neg\emptyset,\ i(k) \cap i(m) = \neg\emptyset,$
  $b(k) \cap i(m) = \neg\emptyset,\ i(k) \cap b(m) = \neg\emptyset$

For spatial objects $so \in SO$, we define the topological relationships already presented in table 3.1 with the prefix *sl*. Let $k,\ m$ be two $so \in SO$ then

- *k.slDisjoint(m):* Returns true if k and m are disjoint, otherwise returns false. k and m are spatially disjoint if
  $b(k) \cap b(m) = \emptyset,\ i(k) \cap i(m) = \emptyset,$
  $b(k) \cap i(m) = \emptyset,\ i(k) \cap b(m) = \emptyset$

- *k.slContain(m):* Returns true if k contains m, otherwise returns false. k spatially contains m if
  $b(k) \cap b(m) = \emptyset,\ i(k) \cap i(m) = \neg\emptyset,$
  $b(k) \cap i(m) = \emptyset,\ i(k) \cap b(m) = \neg\emptyset$

- and similarly for the other topological operators.

Others spatial relationships are the positional or directional relationships presented in table 3.2, and we define them as follows:

- *k.above(m):* Returns true if k is above m, otherwise returns false. k is above m if
  $k.sl(Y).inf \geq m.sl(Y).sup$

- *k.left(m):* Returns true if k is on the left side of m, otherwise returns false. k is on the left side of m if
  $k.sl(X).inf \geq m.sl(X).sup$

- *k.below(m):* Returns true if k is below m, otherwise returns false. k is below m if
  $k.sl(Y).sup \leq m.sl(Y).inf$

- *k.right(m):* Returns true if k is on the right side of m, otherwise returns false. k is on the right side of m if
  $k.sl(X).sup \leq m.sl(X).inf$

- *k.directAbove(m):* Returns true if k is directly above m, otherwise returns false. k is directly above m if
  $k.above(m) \wedge m.sl(X).cover(k.sl(X))$

- *k.directLeft(m):* Returns true if k is directly on the left of m, otherwise returns false. k is directly on the left side of m if
  $k.left(m) \wedge m.sl(Y).cover(k.sl(Y))$

- *k.directBelow(m):* Returns true if k is directly below m, otherwise returns false. k is directly below m if
  $k.below(m) \wedge m.sl(X).cover(k.sl(X))$

- *k.directRight(m):* Returns true if k is directly on the right side of m, otherwise returns false. k is directly on the right side of m if
  $k.right(m) \wedge m.sl(Y).cover(k.sl(Y))$

- *k.aboveLeft(m):* Returns true if k is above and on the left side of m, otherwise returns false. k is above and on the left side of m if
  $k.above(m) \wedge k.left(m)$

- *k.belowLeft(m):* Returns true if k is below and on the left side of m, otherwise returns false. k is below and on the left side of m if
  $k.below(m) \wedge k.left(m)$

- *k.belowRight(m):* Returns true if k is below and on the right side of m, otherwise returns false. k is below and on the right side of m if
  $k.below(m) \wedge k.right(m)$

- *k.aboveRight(m):* Returns true if k is above and on the right side of m, otherwise returns false. k is above and on the right side of m if
  $k.above(m) \wedge k.right(m)$

Also included in the **RTree** class are the specifications of the two variables used in proximity queries named $\Delta$. Examples of use of these two types of spatial queries were also shown in the last chapter. We do not include special operators to deal with *nearest* and *farthest* queries. Operator *between* will be calculated based on the two given mbrs if they are disjoint. For two disjoint mbrs a new mbr is calculated, named $p$, which is the rectangular region between $m$ and $j$ as shown in figure 4.4. The new *between* region depends on the position of the two given mbrs, finding out five possibilities of $m$, $j$ spatial location. The *between* function is based on these possibilities, all rotation or reflection allowed.

For spatial objects, we define the spatial proximity relationships already shown in table 3.3. Let *j, k, m* be three objects in *SO*, then *k.near(m), k.far(m), and*

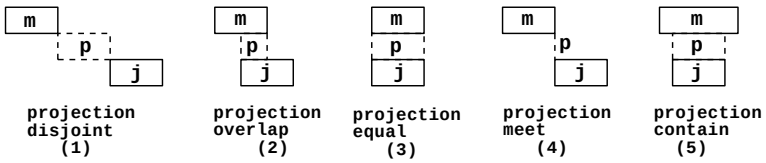



Figure 4.4: New definition of between region.

*k.between(j, m)* are calculated by using topological relationships *k.slOverlap(p), k.slDisjoint(p), and k.slOverlap(p)*, respectively; where $p$ is a new mbr calculated as follow:

- In case of near and far,
  $p = ([m.sl(X).inf - m.ds, m.sl(X).sup + m.ds],$
  $[m.sl(Y).inf - m.ds, m.sl(Y).sup + m.ds])$
- In case of between, the new $p$ depends on the directional relationship between $m$ and $j$.

We support temporal searches where more than one time dimension is involved by applying topological operators to temporal objects in their temporal dimensions. We also include those cases corresponding to temporal searches, where only one dimension of an object is involved, and these relationships were already presented in table 3.4. We define the following temporal operators (top) to be applied to only one time dimension. Let *q, t* be two temporal objects (to) $\in TO$, and *gtd* an integer $gtp \in \mathcal{I}$, $0 < gtd \leq td$, that specifies the given time dimension,

- *q.before(t, gtd):* Returns true if q is before t in that gtd temporal dimension, otherwise returns false. q is before t in gtd if
  $q.tl(gtd) < t.tl(gtd)$
- *q.after(t, gtd)):* Returns true if q is after t in that gtd temporal dimension, otherwise returns false. q is after t in gtd if
  $q.tl(gtd) > t.tl(gtd)$
- *q.during(t, gtd):* Returns true if q is during t in the given temporal dimension, otherwise returns false. q is during t in gtd if
  $b(q.tl(gtd)) \cap b(t.tl(gtd)) = \emptyset,\ i(q.tl(gtd)) \cap i(t.tl(gtd)) = \neg\emptyset,$
  $b(q.tl(gtd)) \cap i(t.tl(gtd)) = \neg\emptyset,\ i(q.tl(gtd)) \cap b(t.tl(gtd)) = \emptyset$
- *q.tMeet(t, gtd):* Returns true if q meets t is the given temporal dimension, otherwise returns false. q meets t in gtd if
  $b(q.tl(gtd)) \cap b(t.tl(gtd)) = \neg\emptyset,\ i(q.tl(gtd)) \cap i(t.tl(gtd)) = \emptyset,$
  $b(q.tl(gtd)) \cap i(t.tl(gtd)) = \emptyset,\ i(q.tl(gtd)) \cap b(t.tl(gtd)) = \emptyset$
- *q.tEqual(t, gtd):* Returns true if q is equal to t in that gtd temporal dimension, otherwise returns false. q is equal to t in gtd if
  $b(q.tl(gtd)) \cap b(t.tl(gtd)) = \neg\emptyset,\ i(q.tl(gtd)) \cap i(t.tl(gtd)) = \neg\emptyset,$
  $b(q.tl(gtd)) \cap i(t.tl(gtd)) = \emptyset,\ i(q.tl(gtd)) \cap b(t.tl(gtd)) = \emptyset$

- *q.start(t, gtd):* Returns true if q and t start on the same point of gtd, otherwise returns false. q and t start equally in gtd if
  $b(q.tl(gtd)) \cap b(t.tl(gtd)) = \neg\emptyset,\ i(q.tl(gtd)) \cap i(t.tl(gtd)) = \neg\emptyset,$
  $b(q.tl(gtd)) \cap i(t.tl(gtd)) = \emptyset,\ i(q.tl(gtd)) \cap b(t.tl(gtd)) = \neg\emptyset$

- *q.finish(t, gtd):* Returns true if q and t finish on the same point of gtd, otherwise returns false. q and t finish equally in gtd if
  $b(q.tl(gtd)) \cap b(t.tl(gtd)) = \neg\emptyset,\ i(q.tl(gtd)) \cap i(t.tl(gtd)) = \neg\emptyset,$
  $b(q.tl(gtd)) \cap i(t.tl(gtd)) = \neg\emptyset,\ i(q.tl(gtd)) \cap b(t.tl(gtd)) = \emptyset$

- *q.adjacent(t, gtd):* Returns true if q is adjacent to t in gtd, otherwise returns false. q is adjacent to t in gtd if
  $t.tl(gtd).inf - q.tl(gtd).sup = o.dt \cup q.tl(gtd).inf - t.tl(gtd).sup = o.dt$

- *q.follow(t, gtd):* Returns true if q is follows t in gtd, otherwise returns false. q follows t in gtd if
  $q.tl(gtd).inf - t.tl(gtd).sup = o.dt$

- *q.precede(t, gtd):* Returns true if q is precedes t in gtd, otherwise returns false. q precedes t in gtd if
  $t.tl(gtd).inf - q.tl(gtd).sup = o.dt$

## 4.3 An object calculus

We also follow the object calculus proposed by M. Tamer, D. Straube, and R. Peters in [TSP93].

A query is of the form {o $\psi$(o)}, where $o$ is an object variable denoting some object in the database and $\psi$ is a formula built from atoms. Atoms represent the primitive query operations of the object model and return a Boolean result. The legal atoms are as follows:

$o_i \theta o_j$ where:

- $o_i$ *and* $o_j$ are object variables or denotes an operation of the form $< o_1 \cdots o_n > .mlist$ where $o_1 \cdots o_n$ are object variables, and $< o_1 \cdots o_n > .mlist$ denotes a multioperation uses when the list of method names is unimportant.
- $\theta$ is one of the operators $==, \in, or =_{\{\}}$

$a \theta o_i$ where:

- $o_i$ is an object variable or denotes an operation of the form $< o_1 \cdots o_n > .mlist$ where $o_1 \cdots o_n$ are object variables.
- $a$ is the textual representation of an atomic value or a set of atomic values.
- $\theta$ is one of the operators $=, \in, or =_{\{\}}$

Formulas depend on the notion of *free* and *bound* variables. A variable is said to be bound in a formula if it has been previously introduced using a quantifier such as $\exists$ or $\forall$. If the variable has not been introduced using a quantifier it is free in the formula.

Formulas are defined as follows:

1. Every atom is a formula. All object variable in the atom are free in the formula.
2. If $\psi_1$ and $\psi_2$ are formulas, then $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, and $\neg\psi_1$ are also formulas. Object variables are free or bound in $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, and $\neg\psi_1$ as they are free or bound in $\psi_1$ or $\psi_2$ depending on where they occur.
3. If $\psi$ is a formula, then $(\exists\ \mathrm{o})(\psi)$ is a formula. Free occurrences of o in $\psi$ are bound to $(\exists\ \mathrm{o})$ in $(\exists\ \mathrm{o})(\psi)$.
4. If $\psi$ is a formula, then $(\forall\ \mathrm{o})(\psi)$ is a formula. Free occurrences of o in $\psi$ are bound to $(\forall\ \mathrm{o})$ in $(\forall\ \mathrm{o})(\psi)$.
5. Formulas may be enclosed in parenthesis. In the absence of parenthesis, the decreasing order of precedence is $\in, =, =_{\{\}}, ==, \exists, \forall, \neg, \wedge$ and $\vee$, in that order.

A query is an object calculus expression of the form $\{\mathrm{o}\ \psi(\mathrm{o})\}$ where $o$ is the only free variable in $\psi$.

We are interested in spatial and/or temporal queries which involve objects in $\mathcal{ST}$ and contain at least one spatial, temporal, or spatio-temporal relationship. The kind of queries we study are expressed in object calculus as follows: {o / o R r} where $o \in \mathcal{ST}$, $r \in HR$, and $R \in STR$. The query means *Find all object o.oid such that o stands in relation R with some given hyper-rectangle r.*

In order to minimize the number of operators in the **RTree** class, a more detailed study must be done. That is, to improve retrieval performances, it is necessary to increment the hit ratio in searches by decreasing the number of disk accesses. This means that search operators must be chosen wisely. We consider the use of the most appropriate relationships between the mbhrs as a constraint for reducing the search space when the query processor is required to execute a spatial and/or temporal query. In the classical R_tree, the only two spatial search operators are match and overlap meaning equality and overlapping of object mbrs. For example, if we have the query {o / o overlaps r}, then we exploit the fact that *o overlaps x* $\Rightarrow$ *mbr(o) overlaps mbr(x)* and rejecting any *o* such that $\neg$*(mbr(o) overlaps mbr(x))*. Thus, the condition *mbr(o) overlaps mbr(x)* is necessary but not sufficient because it is only a one way implication. Using this constraint reduces the number of spatial objects that have to be retrieved, but will nevertheless normally result in some false hits.

In any other cases of range queries, the classical R_tree only provides the overlap search operator.

The R_tree itself comprises a number of nodes such that if node P3 is a child of node P9, then the corresponding entry in P9 covers all of the entries contained in P3 as shown in figure 4.1. To search throughout the tree, we begin the search at the root node by knowing that (P9 cover P3) and (P3 cover x) $\Rightarrow$ (P9 cover x). We make use of this transitivity and a more specific operator in the following way. Assuming that we are trying to answer the query: {o / o cover x} then at some level in the tree, if there is a node such that $\neg$((P9 overlaps x)$\wedge$(P9 cover P3))$\Rightarrow$ $\neg$ (P3 overlaps x), we know we need not search any of the descendants of P9.

For each spatio-temporal operator, we classify as necessary or sufficient the corresponding operator for two *mbhr*s, when the operation is needed for two d-dimensional objects. The necessary column means that the retrieved set of objects after the recommended operation may contain some false hits, and the sufficient column says that the retrieved set of objects does not contain false hits. Table 4.1 shows the specific operator to be called if the spatio-temporal, spatial, and/or temporal query among objects contains any of the operators mentioned above. In the case of spatial queries, the operator used for two polygons is presented in the first column, and the operator invoked for *mbr*s is shown on the right.

The spatial operators support queries of different types, such as region vs. region, region vs. line, region vs. point, line vs. line, line vs. point, and point vs. point. This support is based on the possibility of having the *mbhr* covering the different types of geometrical objects mentioned before. That is, an *mbhr* can be a multidimensional point, if all of its $d$ intervals are intervals of zero length, or a multidimensional line aligned with an axis, if one of its $d$ intervals is an interval of length different from zero, and the others are intervals of zero size. Lines which do not align with an axis must be represented by an *mbhr*.

## 4.4 Extensions of the R_tree searches

The secondary storage access structure which we term **RTree** is intended to support the following operations for a set $B$ of blobs that contains the set $IO$ of indexed objects, organised in a set $E$ of entries composed of the *mbhr(io)* and *bn(io)*, where *bn(io)* is the blob number that contains *io*.

- *Multidimensional searches: stop(RTree(IO), a):* Multidimensional queries are of the form: $\{io/io\ R\ a\}$ where $io \in STO$, $a \in HR$, and $R$ is a binary relationship $\in$

| **d-dimensional object** | **mbhr (dD)** | |
|---|---|---|
| | Necessary | Sufficient |
| disjoint | $\neq$ | disjoint |
| meet | ¬disjoint | - |
| = | = | - |
| cover | cover | - |
| inside | inside | - |
| overlap | ¬disjoint | - |
| **MinSet_dD=**{disjoint,¬disjoint,=,cover,inside} | | |
| **Polygons** | **mbr (2D)** | |
| | Necessary | Sufficient |
| disjoint | $\neq$ | disjoint |
| meet | ¬disjoint | - |
| = | = | - |
| cover | cover | - |
| inside | inside | - |
| overlap | ¬disjoint | - |
| above | left in Y | - |
| below | right in Y | - |
| left | left in X | - |
| right | right in X | - |
| dAbove | left in Y $\wedge$ ¬disjoint in X | - |
| dBelow | right in Y $\wedge$ ¬disjoint in X | - |
| dLeft | left in X $\wedge$ ¬disjoint in Y | - |
| dRight | right in X $\wedge$ ¬disjoint in Y | - |
| aboveLeft | left in X $\wedge$ Y | - |
| belowLeft | right in Y $\wedge$ left in X | - |
| aboveRight | right in X $\wedge$ left in Y | - |
| belowRight | right in X $\wedge$ Y | - |
| **MinSet2D=**{disjoint,¬disjoint,=,cover,inside,leftD,rightD,¬disjointD} | | |
| **Time interval** | **Interval (1D)** | |
| | Necessary | Sufficient |
| disjoint | - | left $\wedge$ right in T |
| before | - | right in T |
| after | - | left in T |
| meet | - | meet |
| = | - | = |
| start | - | start |
| finish | - | finish |
| adjacent | - | adjacent |
| precede | - | precede |
| follow | - | follow |
| during | - | during |
| overlap | - | ¬disjoint in T |
| **MinSet1D=**{rightD,leftD,meet,=,start,finish,adjacent,precede,follow,during,¬disjointD} | | |

Table 4.1: Minimal set of operators for the R_tree index.

$STR$ that contains the topological relationships. Using the information of table 4.1, this query is replaced with $\{io/mbhr(io)\ stop\ a\}$, where *stop* is the corresponding operation in MinSet_dD = {disjoint, nDisjoint, match, cover, inside}. This query returns a list of blob numbers where each $bn \in B$ and $io \in IO$ and this list contains the indexed objects that possibly answer the query.

- *Bidimensional searches: sop(RTree(IO), b):* Bidimensional queries are of the form: $\{io/io\ R\ b\}$ where $io \in SO$, $b \in HR$, and $R$ is a binary relationship $\in SR$ that contains the topological, directional, and proximity relationships. Using the same table, this query is replaced with $\{io/mbhr(io)\ sop\ b\}$, where *sop* is the corresponding operation in MinSet2D={slDisjoint, slnDisjoint, slMatch, slCover, slInside, leftD, rightD, nDisjointD}, and $mbhr(io)$ is defined only in 2D. This query returns the list of blob numbers that contains the indexed objects which possibly answer the query.

- *Unidimensional searches: top(RTree(IO), c):* Unidimensional queries are of the form: $\{io/io\ R\ c\}$ where $io \in TO$, $c \in Int$, and $R$ is a binary relationship $\in TR$ that contains the temporal relationships. Using the same table, this query is replaced with $\{io/mbhr(io)\ top\ c\}$, where *top* is the corresponding operation in MinSet1D={rightD, leftD, meet, tMatch, start, finish, adjacent, precede, follow, during, nDisjointD}, and $mbhr(io)$ is defined only in 1D. This query also returns the list of blob numbers that contains the indexed objects which possibly answer the query.

- *Insert(RTree(IO), e):* Add $e$ to whether the set of entries $E$, and the result is *RTree(IO* $\cup$ $\{io\})$.

- *Delete(RTree(IO), e):* Delete $e$ to the set of entries $E$, if $e$ is in $E$ and the result is *RTree(IO* - $\{io\})$; otherwise RTree remains the same.

At the query language level, a given spatial and/or temporal query can be complex in the sense that it can contain several parts and each of these parts can be catalogued in one of the three kind of query defined above. A given query that calls either a spatio-temporal operator or both spatial and temporal operators is named a spatio-temporal query. To help in this task, we present in table 4.2 our suggested solutions of operator uses. To simplify column identifiers, we use Xi to denote X.inf, Xs to denote X.sup, and so on. The description of the proposed solution to each combination of spatial and temporal operators is joined with the classification of necessary or sufficient, corresponding to the retrieved objects entirely satisfy the given query or not. Spatio-temporal queries are either multidimensional query or a combination of a bidimensional and unidimensional queries. For example, $\{o/o\ disjoint\ a \wedge o\ precede\ b\} \Rightarrow \{o/mbhr(o)\ slDisjoint\ mbhr(a) \wedge o.tl(T)\ precede\ b.tl(t)\}$ where $o \in IO$, $a \in SO$, and $b \in TO$. This spatio-temporal query can be answered by using one multidimensional query as the form: $\{o/mbhr(o)\ disjoint\ mbhr(c)\}$, where $mbhr(c) = ([b.tl(T).inf, HT], mbhr(a))$.

Three proximity operators are also considered in this work as mentioned in the last section. These three operators can be transformed into two of the topological operators. Table 4.3 presents this transformation. The proximity measures, coincidence and neighboring, treated by Roberts in [Rob95] will not be provided by the index.

Combinations of topological, directional, proximity, and temporal operators are possible. To combine directional and proximity, first the proximal operators are transformed to topological, and the result combined with the directional operator. The combination of directional and temporal operators cannot be solved by use of a unique operator, thus it has to be solved separately, and then intersecting the list of retrieved objects. Table 4.4 illustrates the other two cases of combination.

There must exist a query language and a query optimizer to translate user queries to *RTree* class operators. In the following is presented some types of queries that can be formulated by users, together with the *RTree* class operator(s), which can be used to answer those queries, after the intervention of the query optimizer.

1. Queries when it is known exactly where and when

   - Which objects are located at *spatial location* (sl) in this *time interval* ($w$)?
     Expressed as: $\{io$ / *mbhr(io) slMatch* $sl \wedge io.tl(T) = w\} \Rightarrow \{io$ / *mbhr(io)* $= mbhr(c)\}$, where $sl \in HR$ with 2D, $w \in Int$, $io \in IO$, $c \in HR$, and
     $c = ([w.inf, w.sup], [sl.hr(X).inf, sl.hr(X).sup], [sl.hr(Y).inf, sl.hr(Y).sup])$.
     The *RTree* operator *match* is used for this query. Given an instance, *rt*, of the *RTree* class and an instance, $c$, of the *HR* (i.e. hyper-rectangle) class, then query is executed by calling rt.match(c). If the defined time dimension is zero, then $c$ is as stated above. If the defined time dimension is 2, and the spatial dimensions are zero and one, then
     $c = [sl.hr(X).inf, sl.hr(X).sup], [sl.hr(Y).inf, sl.hr(Y).sup], [w.inf, w.sup])$.
   - Which objects do not meet sl in this w? or Which objects are located outside sl in this $w$?
     Query: {io / mbhr(io) slDisjoint sl $\wedge$ io.tl(T) = w} $\Rightarrow$ {io / mbhr(io) disjoint c $\vee$ mbhr(io) disjoint d}, where $sl \in HR$ with 2D, $w \in Int$, $io \in IO$, $c \in HR$, and $c = ([w.inf - io.dt, HT]$,
     $[sl.hr(X).inf, sl.hr(X).sup]$, $[sl.hr(Y).inf, sl.hr(Y).sup])$, and
     $d = ([LT, w.sup + io.dt]$, $[sl.hr(X).inf, sl.hr(X).sup]$,
     $[sl.hr(Y).inf, sl.hr(Y).sup])$.
     Answer will be in: rt.disjoint(c)$\wedge$ rt.disjoint(d).
   - Which objects meet sl in this $w$?
     Query: {io / mbhr(io) slNDisjoint sl $\wedge$ io.tl(T) = w} $\Rightarrow$ {io / mbhr(io) nDisjoint c}, where $sl \in HR$ with 2D, $w \in Int$, $io \in IO$, $c \in HR$, and
     $c = ([w.inf, w.sup], [sl.hr(X).inf, sl.hr(X).sup], [sl.hr(Y).inf, sl.hr(Y).sup])$.
     Answer will be in: rt.nDisjoint(c).

| ([Xi,Xs],[Yi,Ys]) **mbr** | [Ti,Ts] **t** | (S): sufficient (N): necessary | |
|---|---|---|---|
| disjoint | disjoint | disjoint | mbhr=([Ti,Ts],mbr) (S) |
| | before | disjoint | mbhr=([Ti,HT],mbr) (S) |
| | after | disjoint | mbhr=([LT,Ts],mbr) (S) |
| | meet | disjoint | mbhr=([Ti+ΔT,Ts-ΔT],mbr) (N) |
| | = | disjoint | mbhr=([Ti-ΔT,HT],mbr) ∨ mbhr=([LT,Ts+ΔT],mbr) (N) |
| | start | disjoint | mbhr=([LT,Ti],mbr) (N) |
| | finish | disjoint | mbhr=([Ts,HT],mbr) (N) |
| | adjacent | disjoint | mbhr=([Ti,Ts],mbr) (N) |
| | precede | disjoint | mbhr=([Ti,HT],mbr) (N) |
| | follow | disjoint | mbhr=([LT,Ts],mbr) (N) |
| | during | disjoint | mbhr=([LT,Ti-ΔT],mbr) ∨ mbhr=([Ts+ΔT,HT],mbr) (N) |
| | overlap | slDisjoint | mbr (N) |
| ¬disjoint | disjoint | slnDisjoint | mbr (N) |
| | before | ¬disjoint | mbhr=([LT,Ti],mbr) (N) |
| | after | ¬disjoint | mbhr=([Ts,HT],mbr) (N) |
| | meet | ¬disjoint | mbhr=([Ti,Ts],mbr) (N) |
| | = | ¬disjoint | mbhr=([Ti,Ts],mbr) (N) |
| | start | ¬disjoint | mbhr=([Ti,Ti],mbr) (N) |
| | finish | ¬disjoint | mbhr=([Ts,Ts],mbr) (N) |
| | adjacent | ¬disjoint | mbhr=([Ti,Ts],mbr) (N) |
| | precede | ¬disjoint | mbhr=([LT,Ti-ΔT],mbr) (N) |
| | follow | ¬disjoint | mbhr=([Ts+ΔT,HT],mbr) (N) |
| | during | ¬disjoint | mbhr=([Ti,Ts],mbr) (N) |
| | overlap | ¬disjoint | mbhr=([Ti,Ts],mbr) (S) |
| = | disjoint | slMatch | mbr (N) |
| | before | slMatch | mbr (N) |
| | after | slMatch | mbr (N) |
| | meet | slMatch | mbr (N) |
| | = | = | mbhr=([Ti,Ts],mbr) (S) |
| | start | slMatch | mbr (N) |
| | finish | slMatch | mbr (N) |
| | adjacent | slMatch | mbr (N) |
| | precede | slMatch | mbr (N) |
| | follow | slMatch | mbr (N) |
| | during | slMatch | mbr (N) |
| | overlap | slMatch | mbr (N) |
| cover | disjoint | slCover | mbr (N) |
| | before | cover | mbhr=([Ti,Ti],mbr) (N) |
| | after | cover | mbhr=([Ts,Ts],mbr) (N) |
| | meet | slCover | mbr (N) |
| | = | = | mbhr=([Ti,Ts],mbr) (N) |
| | start | cover | mbhr=([Ti,Ts],mbr) (N) |
| | finish | cover | mbhr=([Ts,Ts],mbr) (N) |
| | adjacent | cover | mbhr=([Ti-ΔT,Ti-ΔT],mbr) ∨ mbhr=([Ts+ΔT,Ts+ΔT],mbr) (N) |
| | precede | cover | mbhr=([Ti-ΔT,Ti-ΔT],mbr) (N) |
| | follow | cover | mbhr=([Ts+ΔT,Ts+ΔT],mbr) (N) |
| | during | cover | mbhr=([Ti,Ts],mbr) (S) |
| | overlap | slCover | mbr (N) |
| inside | disjoint | slInside | mbr (N) |
| | before | inside | mbhr=([LT,Ti+ΔT],mbr) (S) |
| | after | inside | mbhr=([Ts-ΔT,HT],mbr) (S) |
| | meet | slInside | mbr (N) |
| | = | inside | mbhr=([Ti-ΔT,Ts+ΔT],mbr) (N) |
| | start | inside | mbhr=([Ti-ΔT,HT],mbr) (N) |
| | finish | inside | mbhr=([LT,Ts+ΔT],mbr) (N) |
| | adjacent | slInside | mbr (N) |
| | precede | inside | mbhr=([LT,Ti],mbr) (N) |
| | follow | inside | mbhr=([Ts,HT],mbr) (N) |
| | during | inside | mbhr=([Ti-ΔT,Ts+ΔT],mbr) (S) |
| | overlap | slInside | mbr (N) |
| LT=lowest time value, HT=highest time value, and ΔT=time unit | | | |

Table 4.2: Combination of topological 2D and temporal 1D operators.

| ([Xi,Xs],[Yi,Ys]) **mbr** | (N): necessary | |
|---|---|---|
| near | ¬disjoint | mbr=(Xi-Δ, Xs+Δ, Yi-Δ, Ys+Δ) (N) |
| far | disjoint | mbr=(Xi-Δ, Xs+Δ, Yi-Δ, Ys+Δ) (N) |
| between | ¬disjoint | mbr=selon redefinition aforementioned (N) |
| Δ is the spatial unit | | |

Table 4.3: Transformation of proximity to topological operators.

| ([Xi,Xs],[Yi,Ys]) **mbr** | | (N): necessary | |
|---|---|---|---|
| disjoint or ¬disjoint | above | above | mbr (N) |
| | below | below | mbr (N) |
| | left | left | mbr (N) |
| | right | right | mbr (N) |
| | dAbove | dAbove | mbr (N) |
| | dBelow | dBelow | mbr (N) |
| | dLeft | dLeft | mbr (N) |
| | dRight | dRight | mbr (N) |
| | aboveLeft | aboveLeft | mbr (N) |
| | belowLeft | belowLeft | mbr (N) |
| | aboveRight | aboveRight | mbr (N) |
| | belowRight | belowRight | mbr (N) |
| Any other combination has not any sense | | | |
| COMBINATION OF TOPOLOGICAL AND DIRECTIONAL OPERATORS. | | | |
| ([Xi,Xs],[Yi,Ys]) **mbr** | | **[Ti,Ts] t** | |
| near far between | | disjoint before after = start finish adjacent precede follow during overlap | After transforming proximity operators to the corresponding topological ones, it is applied the same table for the combination of topological and temporal operators. |
| COMBINATION OF PROXIMITY AND TEMPORAL OPERATORS. | | | |

Table 4.4: Combination of topological, directional, proximity (2D), and temporal 1D operators.

- Which objects cover sl during $w$? or Is sl covered by which objects during $w$?
  Query: {io / mbhr(io) slCover sl $\wedge$ io.tl(T) during w} $\Rightarrow$ {io / mbhr(io) cover c}, where $sl \in HR$ with 2D, $w \in Int$, $io \in IO$, $c \in HR$, and
  $c = ([w.inf, w.sup], [sl.hr(X).inf, sl.hr(X).sup], [sl.hr(Y).inf, sl.hr(Y).sup])$.
  Answer will be in: rt.cover(c).

- Which objects are inside sl in this $w$? or Which objects are contained in sl in this $w$?
  Query: {io / mbhr(io) slInside sl $\wedge$ io.tl(T) $= w$} $\Rightarrow$ {io / mbhr(io) inside c}, where $sl \in HR$ with 2D, $w \in Int$, $io \in IO$, $c \in HR$, and
  $c = ([w.inf - dt, w.sup + dt], [sl.hr(X).inf, sl.hr(X).sup]$,
  $[sl.hr(Y).inf, sl.hr(Y).sup])$.
  Answer will be in: rt.inside(c).

- Which objects overlap sl in this $w$? or Which objects intersect sl in this $w$?
  Query: {io / mbhr(io) slNDisjoint sl $\wedge$ io.tl(T) $= w$} $\Rightarrow$ {io / mbhr(io) nDisjoint c}, where $sl \in HR$ with 2D, $w \in Int$, $io \in IO$, $c \in HR$, and
  $c = ([w.inf, w.sup], [sl.hr(X).inf, sl.hr(X).sup], [sl.hr(Y).inf, sl.hr(Y).sup])$.
  Answer will be in: rt.nDisjoint(c).

- Which objects were located at sl in 1996? Before use the operator, it must be built the time interval, $w$=[19960101,19961231].
  Query: $\{io/mbhr(io)\ slMatch\ sl \wedge io.tl(T) = w\} \Rightarrow \{io/mbhr(io)\ match\ c\}$, where $sl \in HR$ with 2D, $w \in Int$, $io \in IO$, $c \in HR$, and
  $c = ([w.inf, w.sup], [sl.hr(X).inf, sl.hr(X).sup]$,
  $[sl.hr(y).inf, sl.hr(Y).sup])$.
  Answer will be in: rt.match(c).

- The operators mentioned above can be used to restrict the object base to a specific year. Which objects do not meet sl in 19XX? or Which objects are located outside sl during 19XX? or Which objects meet sl after 19XX? or Which objects are covered by sl in 19XX?, and so on.

- The operators can also be used to ask a larger period, for example Which objects were located at sl from 1994 to today? In this case the time interval must be built before any operator is used, for example, w=(19940101,today) if the time granularity is days.

- Temporal operators can also be used. Which objects were located at sl before 1994? In this case, w=[19931231,19931231] if the time granularity is days.
  Query: $\{io/mbhr(io)\ slInside\ sl \wedge io.tl(T)\ before\ w\} \Rightarrow$
  $\{io/mbhr(io)\ inside\ c\}$, where $sl \in HR$ with 2D, $w \in Int$, $io \in IO$, $c \in HR$, and
  $c = ([LT, w.sup + dt], [sl.hr(X).inf, sl.hr(X).sup]$,
  $[sl.hr(Y).inf, sl.hr(Y).sup])$, where LT is the minimum time value in the temporal domain of the attribute.
  Answer will be in: rt.inside(c).

- Direction operators can also be used. Which objects were located above sl after 1994? Here, $w$=[19950101, today] if the time granularity is days.
  Query: $\{io/mbhr(io)\ above\ sl \wedge io.tl(T)\ during\ w\} \Rightarrow$
  $\{io/mbhr(io)\ rightD\ c(Y) \wedge t(io)\ during\ c(T)\}$, where $sl \in HR$ with 2D,

$w \in Int$, $io \in IO$, $c \in HR$, and
$c = ([w.inf, w.sup], [sl.hr(X).inf, sl.hr(X).sup], [sl.hr(Y).inf, sl.hr(Y).sup])$.
Answer will be in: rt.rightD(c, Y) and rt.rightD(c, T).

2. Queries when it is known exactly where. Spatial queries answered by using bidimensional search.

   - Which objects are located at sl? or Which objects are located at sl now?
     Query: $\{io/mbr(io) = sl\} \Rightarrow \{io/mbr(io) = sl\}$, where $sl \in HR$ with 2D, $io \in IO$.
     Answer will be in: rt.slMatch(sl) and it must be choose the retrieved objects with the largest time.
   - For what date is the most recent object located at sl?
     Query: $\{io/mbr(io) = sl\} \Rightarrow \{io/mbr(io) = sl\}$, where $sl \in HR$ with 2D, $io \in IO$.
     Answer will be in: rt.slMatch(sl) and it must be choose the retrieved objects with the smallest time.
   - It can be used the aforementioned operators to ask the object base for an specific location. Which objects are contained in sl now? or Which objects have been contained in sl? or Which objects meet sl now? or Which objects are covered by sl?, and so on.
   - Direction operators can also be used. Which objects were located below of sl?
     Query: $\{io/mbr(io)\ below\ sl\} \Rightarrow \{io/mbr(io)\ leftD\ sl(X)\}$, where $sl \in HR$ with 2D, $io \in IO$.
     Answer will be in: rt.leftD(sl, X).
   - It may be defined two length for expressing near distances and use them to ask the object base. For example, $\Delta$ is used to define mbhr=(w, Xi-$\Delta$, Xs+$\Delta$, Yi-$\Delta$, Ys+$\Delta$) to answer, Which objects are located near sl?
     Query: $\{io/mbr(io)\ near\ sl\} \Rightarrow \{io/mbr(io)\ slNDisjoint\ d\}$, where $sl, d \in HR$ with 2D, $io \in IO$, and
     $d = ([sl(X).inf - ds, sl(X).sup + ds], [sl(Y).inf - ds, sl(Y).sup + ds])$.
     Answer will be in: rt.slNDisjoint(d).
   - With the same d, it is possible to ask Which objects are located far away of sl? rt.slDisjoint(d)

3. Queries when it is known exactly when. Temporal queries answered by using unidimensional search.

   - Which are the existing objects at w?
     Query: $\{io/t(io) = w\} \Rightarrow \{io/t(io) = w\}$, where $io \in IO$, and $w \in Int$.
     Answer will be: rt.equal(w).
   - Which are the existing objects before w?
     Query: $\{io/t(io)\ before\ w\} \Rightarrow \{io/t(io)\ leftDc(T)\}$, where $io \in IO$, $c \in HR$, $w \in Int$,
     $c = ([w.inf, w.sup], [LS, HS], [LS, HS])$, and $LS$, $HS$ are the minimum and the maximum space value in the spatial domain of the attribute, respectively.
     Answer will be in: rt.leftD(c, T).

- It can be used other unidimensional operators. For example, rt.rigthD(c, T).
- Which objects have been created at the beginning of w?
  Query: $\{io/t(io)\ start\ w\} \Rightarrow \{io/t(io)\ start\ w\}$, where $io \in IO$, and $w \in Int$.
  Answer: rt.start(w).

## 4.5 R_tree class requirements

Presented below are the main requirements imposed by the spatio-temporal applications to the index structure. With each requirement, is included whether the implementations can support it or not.

1. Spatial objects are characterized by a geometric component that determines shape and position of the object in space. Temporal objects are those objects that include at least one temporal domain assigned to an attribute.
2. An object's operation may be geometrical such as computation of distance or intersection of polygons; or non-geometrical such as computation of the area of a polygon, insertions, deletions, and so on.
3. The type of expected queries are spatial, temporal, and spatio-temporal.
4. Spatial objects are associated with rectangular regions (mbhr) through their geometric attributes such as location in space.
5. Objects should be saved in large binary blocks named *blobs* of long fixed size for an object base, but blob size may vary from an object base to another.
6. *The blob* minimum occupancy should be guaranteed to be 50%, that is half of the blob must be full.
7. The maximum update time for the insertion or deletion of a single object has to be logarithmic or better in the total number of indexed *blobs*.
8. If the full key of any individual object is specified, the object can always be accessed via direct path from directory to data *blobs*. The access time for one *blob* should be constant and predictable for a given dimension in the index.
9. The index structure should be balanced, multikey, data determined and spatial.
10. All of the index properties have to be preserved under insertion-deletion. It should be fully dynamic.
11. The maximum size of the index should be directly proportional to the quantity of the data.
12. The global order should be preserved. By clustering strategy, objects in nearby *blobs* should be close in space. By buffering strategy, objects can be prefetched before they are needed by exploiting knowledge about structural, positional and inheritance relationships.

13. The spatial distribution of the index regions at each level of the tree should correspond as closely as possible to the spatial distribution at the next lower level, and to that of the *blobs*.

14. The index structure should respect the spatial proximity of objects by distributing objects physically on disk according to their location in space.

15. The representation of empty space should be minimized.

16. The index grows if new objects are inserted in the object base (OB). Growing on demand. The insertion of a new object in a *blob* may cause an *blob* overflow. *Blobs* are selected for splitting when they overflow, thus it is necessary for a new *blob* to be used for selected objects belonging to the overflowing *blob*.

17. Splitting policies have to achieve symmetry in the $d$ dimensions. Each key should be treated as a secondary key. It should use an order partitioning different from cyclic partitioning, which is a blind policy of partitioning that does not take into account the particularities of the current indexed data.

18. The index shrinks if existing objects are deleted in the OB and the percentage of deleted objects is great than a given percentage and the object base manager (OBM) decides to do it. In other words shrinking is performed on demand.

19. *Blobs* are selected for merging when each one of the following conditions holds: two of them are close, one of them underflows, and the sum of the objects in both *blobs* is less or equal than the maximum number of objects in a *blob*.

20. Merging policies have to maintain index properties, and it should be done when the percentage of occupancy in two close *blobs* permit them to be merged.

Some parameters of efficiency can be used for performance study. They are the following:

**ex** number of external storage accesses

**ro** number of retrieved objects (retrieved by the index as a possible answering object)

**ao** number of answering objects (objects that really answer the query)

**n** number of objects

**m** number of *blobs*

1. A parameter proposed by Robinson [Rob81]. $0 \leq \frac{\frac{ao}{n}}{\frac{ex}{m}} < 1$

2. A second parameter used by Nievergelt [NHS84]. $0 \leq \frac{ao}{ro} < 1$

3. The last parameter presented by Seeger [KSSS89] and called the *hit ratio*$=\frac{ao}{c.ex}$, where $c$ is a constant defined in the mentioned reference.

## 4.6 Concluding remarks

The presentation of an object calculus to define the extensions of the R_tree structure is one of the main contribution of this work.

We realize that the object-oriented approach allows us to have an *extensible* R_tree class, which may contain whatever operator or function the applications need, to improve the retrieval performances in a specific domain.

We have also demonstrated that *mbr*s are useful for defining a set of retrieved objects containing the set of answering objects as a first step of object retrieval. We show how to reduce the size of this set of retrieved objects, i.e. augment the hit ratio, by focusing on the use of more specific search operators instead of using a more complicated object shape container.

Finally, the presentation and formal definition of the minimal set of spatial, temporal, and spatio-temporal operators, is the second main contribution of this work.

# Chapter 5

# R_tree design and implementation

This chapter presents the object-oriented design and several implementations of this structure. The design of the **RTree** class is made based on the Técnica de Desarrollo de Sistemas de Objetos (TDSO) technique [Bes95], and the implementations were done in the C++ language under the operating system Irix version 5.3 running on a standalone Silicon Graphics machine.

TDSO is an object-oriented technique that integrates the concepts of the deductive method (MEDEE) [Duf88] and the object modeling technique (OMT) [RBP+91]. TDSO contains the main concepts to develop object-oriented software following a step-by-step approach. It permits a formal specification of abstract data types (ADT) and a structured design of software systems. The most important concept in TDSO is the inclusion in the design step of the test cases specification. For that, the function specification forms of TDSO contain the algorithm of the function, and its test cases, both accompanied by the documentation needed for each variable, function, and test case.

This chapter is organised as follows: Section 5.1 shows the principal forms of TDSO and how they are used to specify the **RTree** class and its basic classes. The five extensions made of the R_tree structure that expand the range search function for several more specific range search functions depending on topological, directional, or proximity searches, and a new structure called O_tree are described in section 5.2, and finally, section 5.3 contains some concluding remarks.

## 5.1 Specification of the R_tree classes

The specification of the object-oriented classes utilized to implement the R_tree index structure is presented in this section by following the TDSO technique presented by the author of this thesis in [Bes95]. TDSO is a new technique to develop object-oriented software. This technique supports data abstraction and object-oriented concepts. It is based on the deductive method (MEDEE) proposed by J. Dufourd in [Duf88], its extension proposed by I. Besembel in [Bes94], and the methodology OMT proposed by Rumbaugh et al. in [RBP+91]. It uses algebraic specification proposed by Guttag in [Gut77] and [GHM78], for each abstract data type (ADT). The specification of an ADT is composed of *syntactic specification* that defines names, domains, and ranges of each operation or function defined for this ADT; and *sample semantics* that contain a **non complete** set of axioms specifying the behaviour of a function in extreme or bounded cases. A correct implementation of an ADT must satisfy its specification, but it can contain a larger set of operations than those specified for the ADT. An ADT can be implemented in several ways, each implementation is composed of a structure and algorithms. The *structure* defines its data structure, and the *algorithms* state how this data structure will be used and manipulated.

In TDSO, the class hierarchy associated to the is-a relationship is represented by double arrows, and the composition hierarchy associated to the part-of relationship is represented by single arrows. Figure 5.1 shows the composition hierarchy of the R_tree index.

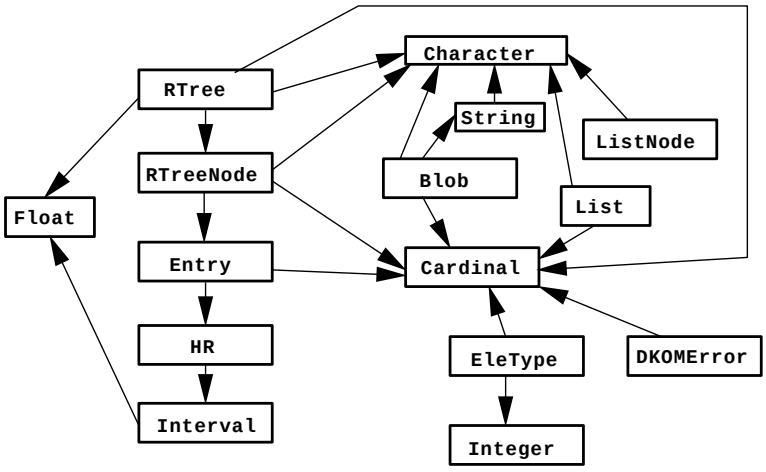



Figure 5.1: TDSO diagram of the composition hierarchy of the R_tree index.

The specification of classes of a system in TDSO is presented in a form named the **Universe** of the system, this specification is described in figure 5.2, corresponding to

the R_tree system. It contains all types and classes used in the system in question. The universe form shows a numeration of types and classes, and each of them is documented in the third column of the form.

The R_tree structure can be formally specified as an ADT and implemented as an object-oriented class. Figure 5.3 shows the seventeenth class of the universe, the **RTree** class that has seven basic operations which are self documented in the TDSO specification form. This form contains the date of the specification, the number associated with the class, the set of related classes, and three columns containing the list of each part of the specification, the specification itself, and the documentation of each ADT operation. These operations conform to the minimal set of operations needed to define the ADT. In the ADT implementation, this set can be expanded depending on the language used, and on the needs of the application.

According to L. Valet in [Val96], object-oriented operations are categorized as constructors, mutators, observers, and destructors. A *constructor* function creates and initiates objects of that class. *Mutators* change the values stored in the object promoting a change of the state of the object. *Observers* simply display or return the value(s) stored in the object without changing any value at all. Finally, a *destructor* destroys the object. Each documentation made in a TDSO specification form follows this categorization for operations.

The rest of the classes mentioned in the universe of the R_tree index are presented in the following figures.

Figure 5.4 shows the specification of the **Interval** class, which is the twelfth class in the R_tree index universe. This class contains six basic operations, which support the minimal set of operations for an ADT. Each valid interval object has two values and the superior value has to be greater than or equal to the inferior one. An emptyInterval is one whose two values are the same.

Figure 5.5 presents the **HR** (hyper-rectangle) class, the thirteenth class in the R_tree index universe. This class is composed of an array of objects of the Interval class. The size of the array depends on the number of dimensions defined in the R_tree index. The dimensions used in these classes are assumed to be homogeneous.

Figure 5.6 describes the ninetieth class of the universe, the **ListOf** class, which is a parameterized class because its elements can be of any type already defined in the system. A parameterized class is a special kind of class that is defined to manage a class of any other class or type of object. The required class is passed to the parameterized class instead of the generic type T. This class will be implemented using a C++ template. In the C++ language, a template permits definition of

| Apr.96 | **Universe R_tree index** | | |
|---|---|---|---|
| 1 | Character | | -**Character**: Type: Basic type. |
| 2 | Integer | | -**Integer**: Type: Basic type. |
| 3 | Float | | -**Float**: Type: Basic type. |
| 4 | Pointer | | -**Pointer**: Type: Basic type. |
| 5 | Cardinal: | Integer+ | -**Cardinal**: Type: Subset of Integer, |
| 6 | String: | ArrayOf Character | only positive values. Basic type. |
| 7 | ListNode: | **struct** | -**String**: Type: Basic type. |
| | | item: Pointer | -**ListNode**: Class: Generic list node. |
| | | to Character | -**List**: Class: List descriptor. |
| | | prior: Pointer | -**ListOf[T]**: Parameterized class: |
| | | to Character | Define a list of elements of |
| | | next: Pointer | any type already defined. |
| | | to Character | -**DKOMError**: Class: Simple class to |
| | | **endstruct** | manage errors in the index. |
| 8 | List: | **struct** | -**EleType**: Class: Simple class to |
| | | n: Cardinal | define the elements of a list of |
| | | pos: Cardinal | RTreeNode numbers and its position |
| | | length: Cardinal | into the RTreeNode father. It |
| | | head: Pointer | simulate a stack containing the |
| | | to Character | descending path of the tree. |
| | ListOf[T] | tail: Pointer | -**Interval**: Class: Define a bounded |
| | | to Character | interval of real numbers. |
| | | actual: Pointer | -**HR**: Class: Define a hyper-rectangle |
| | | to Character | of d-dimensions as an ArrayOf Interval. |
| | | **endstruct** | -**Blob**: Class: Management of |
| 10 | DKOMError: | **struct** | blobs that can contain objects of any |
| | | c: Cardinal | class. It is used to store or retrieve |
| | | **endstruct** | the R_tree nodes and the information |
| 11 | EleType: | **struct** | of the R_tree object on disk. |
| | | pn: Cardinal | -**Entry**: Class: Define the entries |
| | | po: Integer | of the R_tree nodes. |
| | | **endstruct** | -**RTreeNode**: Class: Define the nodes |
| 12 | Interval | | that conform an R_tree index. |
| 13 | HR | | -**RTree**: Class: Define the R_tree index. |
| 14 | Blob | | -**MPoint**: Class: Define a |
| 15 | Entry | | multi-dimensional point used |
| 16 | RTreeNode | | in the multipoint R_tree. |
| 17 | RTree | | -**LeafEntry**: Class: Define |
| 18 | MPoint | | the leaf entries contained |
| 19 | LeafEntry | | in leaf nodes.(multipoint only) |

Figure 5.2: Universe of the **R_tree index**.

| Apr.96 | **Specification {17}Class RTree** Classes: Boolean, Entry, HR, Integer, ListOf | |
|---|---|---|
| 1 | *Syntactic:* | **-createRTree()**: |
| | createRTree()→RTree, | Constructor and initiator. |
| | insert(RTree,Entry)→RTree, | Create an empty R_tree. |
| | delete(RTree,Entry)→RTree, | (emptyRTree). |
| | search(RTree,HR)→ListOf Integer, | **-insert()**: Mutator. |
| | cleanRTree(RTree)→RTree, | Insert a new entry. |
| | isEmpty(RTree)→Boolean, | **-delete()**: Mutator. |
| | destroyRTree(RTree)→. | Delete an existing entry. |
| 2 | *Variables:* | **-search()**: Observer. |
| | RTree: rt, emptyRTree | Return a list of blobId |
| | Entry: e | corresponding to entries |
| | HR: hr ListOf Integer: emptyList | that answer the search. |
| 3 | *Sample semantics:* | **-cleanRTree()**: Mutator. |
| | isEmpty(createRTree())=True | Returns an empty R_tree. |
| | isEmpty(insertEnt(createRTree(),e))=False | **-isEmpty()**: Observer. |
| | isEmpty(deleteEnt(createRTree(),e))=True | Return true if the R_tree |
| | search(createRTree(),hr)=emptyList | is empty, otherwise false. |
| | isEmpty(cleanRTree(rt))=True | **-destroyRTree()**: De- |
| | search(cleanRTree(rt),hr)=emptyList | structor. Destroy the tree |
| | delete(insert(createRTree(),e),e)=emptyRTree | after writing it on disk. |

Figure 5.3: TDSO specification of the **RTree** class.

generic classes, which contain generalizable structure and behaviour for any type of object indicated by T. Because C++ is not a complete object-oriented language, the manner for supporting parameterized classes is by generation of code corresponding to a duplication of the defined code of type T, but now with the specific type passed as a template parameter.

Figure 5.7 illustrates the fourteenth class of the R_tree index universe, called the **Blob** class. The specification required for the R_tree index only supports facilities for reading and writing R_tree nodes, and **RTree** objects on disk. Others functionalities of this class, such as package of objects for storage and retrieval, are not presented here. The complete specification of this class is presented by M. Jorge in [Jor94], and it implements the first version of the object storage manager for the D/K model. Since the indexed objects, others than R_tree nodes and R_tree objects, can be packaged into a blob (several indexed objects into a blob), the blob number (bn) associated with each leaf entry in the R_tree index may be repeated. However, the bn in branch nodes must be unique.

| Apr.96 | **Specification {12}Class Interval**<br>Classes: Float, Boolean, Cardinal | |
|---|---|---|
| 1 | *Syntactic:*<br>createInterval()→Interval,<br>change(Interval,Float,Boolean)→Interval,<br>retrieve(Interval,Boolean)→Float,<br>compare(Interval,Interval)→Boolean,<br>assign(Interval,Interval)→Interval,<br>extent(Interval)→Cardinal,<br>isAPoint(Interval)→Boolean,<br>isEmpty(Interval)→Boolean,<br>destroyInterval(Interval)→. | **-createInterval()**:<br>Constructor e initiator.<br>Create an emptyInterval.<br>**-change()**: Mutator.<br>If the second parameter<br>is true, inferior is changed,<br>otherwise superior changes.<br>**-assign()**: Mutator. Interval<br>assignment.<br>**-isEmpty()**: Observer. Returns<br>true if both values are null.<br>**-retrieve()**: Observer. If para-<br>meter is true, returns the inferior,<br>otherwise the superior value.<br>**-compare()**: Observer.<br>Return true if both intervals<br>satisfy the comparison operator.<br>**-isAPoint()**: Observer.<br>Returns true if the interval<br>is a point, otherwise false.<br>**-extent()**: Observer. Returns<br>the extent of the interval<br>which is superior-inferior.<br>**-destroyInterval()**:Destructor.<br>Destroy the interval. |
| 2 | *Variables:*<br>Interval: i, emptyList<br>Float: v<br>Boolean: f | |
| 3 | *Sample semantics:*<br>isAPoint(createInterval())=True<br>isAPoint(change(createInterval(),v,f))=<br>if(v=0) True; else False<br>retrieve(createInterval(),f)=0<br>assign(createInterval(),i)=i<br>assign(i,createInterval())=emptyInterval<br>extent(createInterval())=0<br>retrieve(change(i,v,f),f)=v<br>isEmpty(createInterval())=True | |

Figure 5.4: TDSO specification of the **Interval** class.

Figure 5.8 presents the fifth class of the R_tree index universe, named the **Entry** class. It permits the definition of the structure and behaviour of the entries of the R_tree node. In this specification, only the minimal subset of the methods used to express the behaviour of the class is shown.

Finally, the **RTreeNode** class, the sixteenth class of the R_tree index universe is presented in figure 5.9. This RTreeNode specification is used for leaf and branch nodes. These nodes are read and written on disk by using **blobs** as containers and by calling readNode and writeNode methods.

The corresponding TDSO forms of the implementations of each extension of the **RTree** class are presented in the following subsections. The TDSO implementation forms of the support classes of the R_tree index are contained in appendix B.

| Apr.96 | **Specification {13}Class HR**<br>Classes: Boolean, Cardinal, Interval | |
|---|---|---|
| 1 | *Syntactic:*<br>createHR()→HR,<br>change(HR,Interval,Cardinal)→HR<br>retrieve(HR,Cardinal)→Interval,<br>assign(HR,HR)→HR,<br>compare(HR,HR)→Boolean,<br>volume(HR)→Cardinal,<br>isAPoint(HR)→Boolean,<br>destroyHR(HR)→. | -**createHR()**: Constructor and initiator. Create an empty hyper-rectangle (emptyHR).<br>-**change()**: Mutator. Interval in the given dimension is changed.<br>-**assign()**: Mutator. Hyper-rectangle assignment.<br>-**retrieve()**: Observer. Return the interval of the specified dimension. |
| 2 | *Variables:*<br>HR: h, emptyHR<br>Interval: i, emptyInterval<br>Cardinal: d | -**compare()**: Observer. Return true if both hyper-rectangles satisfy the comparison operator,otherwise false. |
| 3 | *Sample semantics:*<br>isAPoint(createHR())=True<br>isAPoint(change(createHR(),i,d))=<br>if(isEmpty(i)) True; else False<br>retrieve(createHR(),d)=emptyInterval<br>volume(createHR())=0<br>assign(createHR(),h)=h<br>assign(h,createHR())=emptyHR<br>retrieve(change(h,i,d),d)=i | -**volume()**: Observer. Return the volume of the hyper-rectangle. Calculated as a 3D volume.<br>-**isAPoint()**: Observer. Return true if the hyper-rectangle is a point, otherwise false.<br>-**destroyHR()**: Destructor. Destroy the hyper-rectangle. |

Figure 5.5: TDSO specification of the **HR** (hyper-rectangle) class.

## 5.2 The R_tree extensions

As a first step, we study and implement the R_tree structure to augment our knowledge and we found that, in the Guttman's R_tree, it was possible to make some changes to improve its performance.

The first change was the entry deletion problem, because Guttman's paper proposes use of reinsertion of some entries that were in the wrong place after an entry deletion. Being in the wrong place means that these entries have to be moved to another nodes according to the R_tree definition, sometimes resulting in an underfull node which is solved by the merge process, similar to the same process in B_trees. The reinsertion alternative, in our view, is not a good alternative because it may cause a major restructuring of the tree followed by a larger entry deletion time. The second change was the splitting policy, which we found possible to improve by delaying node splitting by using the same redistribution policy used in entry deletion.

The third aspect was introduced after consideration as to whether it might be more appropriate to use specific search operators rather than the very general operator used by Guttman. We found an answer in Egenhofer's topological operators [Ege89], [EH90], [NHM92] and in the position operators. Finally, a new structure based on the R_tree and called the octagon tree (O_tree) is described in the last subsection. All of these extensions were implemented based on the specification done in the last section.

### 5.2.1 The R_tree extension 1

As we mentioned before, this is the first implementation undertaken following Guttman's paper. It is basically the original R_tree but avoiding the reinsertion of entries in the tree after a deletion of a node. In the deletion algorithm, we realize that the policy of reinsertion proposed by the author provoked several possible insertions of old R_tree entries again. In order to avoid this situation, a deletion operation was implemented following a policy of redistribution of entries between the two possible brothers of an underfull R_tree node. As a consequence of this, two references were included in the head of a node, which are blobId of the anterior node (left brother) and blobId of the posterior one (right brother). This policy was maintained for the rest of the extensions. To delete an entry, two functions are provided that redistribute the entries between its left and/or right brother nodes. If the node has only one brother, the redistribution is made with that node and this function is called **redistribution2**. Otherwise, the **redistribution** function is invoked. Figure 5.10 illustrates the situation when the redistribution is made with three nodes, and figure 5.11 shows the situation with two nodes. Empty space within nodes is emphasized with hatching.

The **redistribution** function is used when the number of entries in the underflowed node and those of its brother nodes divided by 3 is greater than $m$. More clearly, let nA be the number of entries of the underflowed node, nB the number of entries of the left brother node, and nCB the number of entries of the right brother node. If $\lceil (nA + nB + nC)/3 \rceil \geq m$, then the **redistribution** function is invoked from the **condenseTree** function, otherwise a test is made to see if a node deletion is required. If $\lceil (nA + nB + nC)/2 \rceil < M$, then the node deletion is done, otherwise a redistribution among two nodes is done (**redistribution2**). This means that a node deletion is done only in the case of the total number of entries among three nodes being insufficient to satisfy the needs of having one of these nodes. After a

redistribution of three nodes, the number of entries among these nodes is almost the same. The consequences of a redistribution is the change of the mbhrs in the corresponding entries of the father nodes.

The **deletion** function empties the underfull node by putting its entries into both brother nodes, and then deletes this node changing the corresponding entries in the father node. After a node deletion, the father node can be underfull and the process may be restarted again, sometimes until reaching the root. In the case of a root deletion, the R_tree shrinks in height and in width. Each time a node changes by one of these functions, the change is reflected on disk, that is each changed node is written to disk.

The **redistribution2** function is called either when $\lceil (nA + nB + nC)/2 \rceil \geq M$, or when the underflowed node only has one brother node, that can be the left or the right one. After the **redistribution2** function is executed, the number of entries in the corresponding nodes is almost the same, and its corresponding entries in its father node are updated.

If the underfull node only has one brother and $\lceil (nA + nb)/2 \rceil < M$, then the **deletion2** function is called. This last function does the same basic operations as the **deletion** function, but with only two nodes. It deletes the underflowed node after passing its entries to its brother, and updating the corresponding entries in the father node. After a node deletion, the father node can be underfull and the process is repeated again at this level, with the possibility of reaching the root node.

The implementation of the R_tree extension 1 class is named **RTreeExt1**, and it uses the same basic classes already specified in the above section. The implementation of these classes are described in appendix B.

### 5.2.2 The R_tree extension 2

Since a B_tree has similar conditions concerning redistribution during insertion and deletion, we decided to make a second R_tree extension that supports these two types of redistribution. This extension is based on extension 1, and it supports redistribution during entry deletion as well as during the insertion of a new entry. Thus, the main difference between both extensions is the insertion function, that delays as much as possible a node splitting. Extension 2 splits an overflowed node only if it is impossible to redistribute extra entries among brothers, and it merges an underflowed node under similar conditions. The consequence is the delay of both split and merge operations in the structure. When a new entry must be inserted

in the structure, the search is made for a leaf node where the insertion must be done. If the leaf node is full, its brother nodes are read, if any. If its brother nodes have enough space to hold new entries, then the redistribution process is done. This redistribution operation sums the number of the entries of the nodes involved in the process, and this result is divided by the number of nodes involved, to give the number of the entries that each node involved will contain. Figure 5.12 shows a diagram of the situation where P is the parent node of nodes A, B, and C. The overflowed node is A. If $\lceil (nA + nB + nC)/3 \rceil \geq m$, then entries are redistributed, and the process is similar to those described to extension 1, except that instead of calling the **deletion** function, the **splitNode** function is called.

If node A has only one brother, which is the case when A is the node on the extreme left or extreme right, then the calculation of the possible redistribution is made with two nodes. This situation can be observed in figure 5.13 where the empty space within each node is emphasized with small hatching.

The corresponding TDSO form to implement this extension is the same as extension 1, because the only difference is a new function, called **overflow**, that is invoked from the **insertEnt** function. This function is charged to verify the splitting conditions, and its signature is **overflow**(RTreeNode, ListOf[EleType]). If a node must be split, the **splitNode** function is called, otherwise the overflowed node is passed to either redistribution or redistribution2 functions, according to the number of brothers of this node. The implementations of its basic classes remain the same as those shown in appendix B.

### 5.2.3 The R_tree extension 3

The third implementation of the R_tree index specification is the **RTreeExt3** class. This class is based on the **RTreeExt1** class with the addition of the spatial and temporal operators already described in chapter 3. The TDSO implementation form of this extension and the corresponding basic classes are presented in appendix B.

This extension permits indexing of spatial objects with more than two dimensions by indicating at the beginning the number of coordinates in **nCoord**, and the position of the first coordinate in **xDim**. It is assumed that the Y coordinate and the following ones, if any, are stored after **xDim**, that is the position of the Y coordinate is **xDim+1**, the following is **xDim+2**, and so on.

It is also possible to use the index with temporal objects by establishing the number of time dimensions in **nTime**, and the position of the first time dimension

in **timeDim**. Once **nTime** and **timeDim** are fixed, it is also possible to set the unit of time needed in **timeUnit**.

When these attributes are different to the system default values, it is established that the R_tree index is a spatio-temporal index, and all the defined operators are available. Thus, spatio-temporal queries can use the set of spatio-temporal operators (stop), i.e. **stop={disjoint, nDisjoint, match, inside, cover}**. If the query is only spatial, then only the set of spatial operators (sop) are available, i.e. **sop={slDisjoint, slnDisjoint, slMatch, slInside, slCover, leftD, rightD, nDisjointD}**. Finally, if the query is only temporal, then only the set of temporal operators (top) may be used, i.e. **top={leftD, rightD, during, equal, adjacent, follow, precede, meet, start, finish, nDisjointD}**.

### 5.2.4 The R_tree extension 4

This implementation is basically the same as extension 3, but it allows redistribution of entries in case of insertion. Thus, this extension delays node splitting, and node deletion processes as much as possible. The other operators provided are the same as for extension 3.

The TDSO implementation form is similar to those of extension 3, except for the inclusion of the **overflow** function. The rest of the classes are kept. Appendix B contains the mentioned TDSO forms.

### 5.2.5 The multi-point R_tree

This implementation is based on extension 3, and permits a better use of the R_tree leaf nodes when it is known that the application only treats multipoint data. That is, if the data contains spatial location and/or time expressed as point values. This is the case for the available REM data that will be explained in chapter 5.

In this extension, the format of leaf and branch nodes are different in the content to the rest of the R_tree nodes. Branch nodes have the same format as described in section one and illustrated in figure 4.2. The leaf node format is changed to that shown in figure 5.14.

Both, leaf and branch nodes are stored in blobs, so the **Blob** class does not change. The new TDSO diagram of the class and composition hierarchies is presented in figure 5.15, where three new classes are added. These classes support the management of data points. We keep the R_tree index universe of figure 5.2, which defines the relevant classes, and the new specification of the **RTree** class is shown

in figure 5.16. We called this extension the multi-point R_tree (**MRTree**) class to differentiate it from the previous class which is more general.

The TDSO specification of the class **MPoint** is presented in figure 5.17. This is the eighteenth class of the R_tree universe, and it is composed of an array of **Float** values. The size of this array depends on the number of dimensions defined for the **MRTree** class. The **MPoint** class defines a point in d-dimensional space.

Figure 5.18 presents the TDSO specification of the **LeafEntry** class, which is the nineteenth class of the R_tree universe. This class is similar to the **Entry** class, except that the attribute **hr** is changed to **mp** pertaining to the **MPoint** class.

Leaf entries are only used in leaf nodes, so the new **RTreeNode** class is specified in figure 5.19. The temporal approach that is supported by this extension is the point model, where points in space and time are stored in leaf nodes. This extension supports the same operators as extension 3.

For the implementation of the **MRTree** class, the insertion and deletion of indexed objects are redefined to insert and delete leaf entries, because hyper-rectangles are only permitted in branch nodes. The TDSO implementation forms of the **MRTree** class and its basic classes, containing the structure and behaviour of the class, and also a brief description of each variable and operation, are contained in appendix B.

### 5.2.6 The O_tree

In order to reduce the hit ratio of searches of the indexed polygonal objects, Roberts proposes a new spatial indexing method, called the O_tree. This tree is an extension of the R_tree structure in 2D, where the *mbr* of the R_tree entries are replaced by two *mbr*s, which together define an 8-sided polygon (octagon).

The first *mbr* is defined as normal; parallel to the X-Y axes. The second one is defined parallel to the X-Y axes rotated through 45°. An O_tree entry contains these two *mbr*s defined as follows: let (X,Y)∈P where P is the set of points defining the polygonal object. The first *mbr*, $mbr_1 : [Xi, Xs, Yi, Ys]$, is calculated with

$$Xi = min_P(X); Xs = max_P(X); Yi = min_P(Y); Ys = max_P(Y)$$

and the second one, $mbr_2 : [C_1, C_2, C_3, C_4]$ is calculated with

$C_1 = min_P(X + Y); C_2 = max_P(X + Y); C_3 = min_P(Y - X); C_4 = max_P(Y - X)$

Each $C_i$ is the value where the sides of $mbr_2$ (extended if necessary) intersect the Y-axis.

Figure 5.20 illustrates the two *mbr*s over a polygonal object, which is shown in

grey. The $C_i$ values used to store $mbr_2$ are also shown.

The structure composed by these two *mbr*s is called a tight-bounding octagon (tbo) and they are included in an O_tree entry with the blob number of the indexed object. To calculate the new tbo that encloses two or more tbos, we use the same formula utilized to calculate the *mbr* that encloses two or more *mbr*s. That is, given two tbos:

$$tbo_1 : [Xi_1, Xs_1, Yi_1, Ys_1, C_{11}, C_{21}, C_{31}, C_{41}], \text{ and}$$
$$tbo_2 : [Xi_2, Xs_2, Yi_2, Ys_2, C_{12}, C_{22}, C_{32}, C_{42}]$$

the tbo enclosing both tbos is given by:

$$[min(Xi_1, Xi_2), max(Xs_1, Xs_2), min(Yi_1, Yi_2), max(Ys_1, Ys_2),$$
$$min(C_{11}, C_{12}), max(C_{21}, C_{22}), min(C_{31}, C_{32}), max(C_{41}, C_{42})]$$

This formula is used to form a new tbo in branch nodes, when a new entry is inserted in the O_tree. Also needed is a new condition to test if two tbos intersect, which is used in searches. Thus, two tbos intersect if both their pairs of *mbr*s overlap. In testing for coverage, as used in exact match queries, it is only needed to test the first *mbr* coverage, because the second *mbr* is superfluous in this test.

The **OTree** class is specified similarly to the **RTree** class. We do not need to change anything in the TDSO specification, so figure 5.3 describes it. The **HR** class is utilized with double the number of dimensions, and the rest of the basic classes remain the same, except for the **RTreeNode** class, where the **maxHb** function is replaced by the **maxTbo** function.

The TDSO implementation form of the **OTree** class is also similar to those of the **RTreeExt3** class, that is described in appendix B.

## 5.3 Concluding remarks

The design of object-oriented classes allows multiple implementations supporting an ADT. Object-oriented languages are powerful tools that permit code reutilization and system extensibility as their main advantages used in this work. The R_tree implementations reuse the basic classes such as **Interval, HR, and Entry**, and these classes were extended over the course of the implementation to cover the needs of each new extension made.

We have shown that the object-oriented approach allows us to have an *extensible* R_tree class, which may contain whatever operator or function the applications need, to improve the retrieval performances in a specific domain.

Finally, the O_tree uses an octagon container instead of a rectangle, but we can-

not avoid the use of the necessary computational geometry which is always needed to obtain the set of answering objects.

| Apr.96 | **Specification {9}Class ListOf[T]** Classes: Cardinal, T | |
|---|---|---|
| 1 | *Syntactic:* | **-createList()**: Constructor. |
| | createList()→ListOf, | Create an empty list |
| | insert(ListOf,T)→ListOf, | of element of type T. |
| | delete(ListOf)→ListOf, | **-insert()**: Mutator. Insert a new |
| | actual(ListOf)→T, | element before the actual element. |
| | next(ListOf)→T, | **-delete()**: Mutator. Delete |
| | prior(ListOf)→T, | the actual element, if any; |
| | cleanList(ListOf)→ListOf, | otherwise emptyList. |
| | numList(ListOf)→Cardinal, | **-actual()**: Observer. Return |
| | destroyList(ListOf)→. | the actual element, if any; |
| 2 | *Variables:* | otherwise emptyT. |
| | ListOf: l, emptyList | **-next()**: Observer. Return |
| | T: t, emptyT | the next element (after |
| 3 | *Sample semantics:* | the actual) and move actual to |
| | numList(createList())=0 | it, if any; otherwise emptyT. |
| | numList(insert(createList(),t))=1 | **-prior()**: Observer. Return |
| | actual(insert(createList(),t))=t | the prior element (before |
| | next(insert(createList(),t))=emptyT | the actual) and move actual to |
| | prior(insert(createList(),t))=emptyT | it, if any; otherwise emptyT. |
| | actual(createList())=emptyT | **-cleanList():** Mutator. |
| | numList(cleanList(l))=0 | Delete all nodes into the |
| | delete(cleanList(l))= emptyList | list and return emptyList. |
| | delete(insert(createList(),t))=emptyList | **-numList()**: Observer. |
| | | Return the current number |
| | | of elements into the list. |
| | | An empty list has zero elements. |
| | | **-destroyList()**: Destructor. |
| | | Clean the list before destroying it. |

Figure 5.6: TDSO specification of the **ListOf** class.

| Apr.96 | **Specification {14}Class Blob**<br>Classes: Character, Boolean, Cardinal, String | |
|---|---|---|
| 1 | *Syntactic:*<br>createBlob()→Blob,<br>getNew(Blob,Cardinal)→Cardinal,<br>assign(Blob,Blob)→Blob,<br>readBlob(Blob,Cardinal,Cardinal)→Blob,<br>writeBlob(Blob,Cardinal,Cardinal)→Boolean,<br>status(Blob,Character)→Blob,<br>objNum(Blob,Character)→Blob,<br>length(Blob,Cardinal)→Blob,<br>left(Blob,Cardinal)→Blob,<br>right(Blob,Cardinal)→Blob,<br>tail(Blob,String)→Blob,<br>gStatus(Blob)→Character,<br>gObjNum(Blob)→Character,<br>gLength(Blob)→Cardinal,<br>gLeft(Blob)→Cardinal,<br>gRight(Blob)→Cardinal,<br>gTail(Blob))→String,<br>destroyBlob(Blob)→. | **-createBlob()**: Constructor and initiator. Create an empty blob in main memory (emptyBlob).<br>**-getNew()**: Observer. Obtain a new blob number from disk.<br>**-assign()**: Mutator. Change the first blob values.<br>**-readBlob()**: Mutator. Read the blob content from disk.<br>**-writeBlob()**: Mutator. Write the blob content on disk.<br>**-status()**: Mutator. Change the status of the blob.<br>**-objNum()**: Mutator. Change its number of the contained objects.<br>**-length()**: Mutator. Change its current length in bytes.<br>**-left()**: Mutator. Change the blob number on its left.<br>**-right()**: Mutator. Change the blob number on its right.<br>**-tail()**: Mutator. Change the tail of the blob.<br>**-gStatus(),gObjNum(), gLength(),gLeft(), gRight(),gTail()**: Observers. Return each value of the attributes of a blob.<br>**-destroyBlob()**: Destructor. Destroy the blob in main memory. |
| 2 | *Variables:*<br>Blob: b, emptyBlob; Character: ch, on<br>Cardinal: c, bId, l; String: s, emptyString | |
| 3 | *Sample semantics:*<br>gStatus(createBlob())=0<br>gObjNum(createBlob())=0<br>gLength(createBlob())=0<br>gLeft(createBlob())=0<br>gRight(createBlob())=0<br>gTail(createBlob())=emptyString<br>readBlob(createBlob(),c,0)=emptyBlob<br>writeBlob(createBlob(),c,0)=False<br>gStatus(status(b,ch))=ch<br>gObjNum(objNum(b,on))=on<br>gLength(length(b,l))=l<br>gLeft(left(b,bId))=bId<br>gRigth(right(b,bId))=bId<br>gTail(tail(b,s))=s<br>assign(createBlob(),b)=b<br>assign(b,createBlob())=emptyBlob | |

Figure 5.7: TDSO specification of the **Blob** class.

| Apr.96 | **Specification {15}Class Entry**<br>Classes: Boolean, Cardinal, Interval, HR | |
|---|---|---|
| 1 | *Syntactic:*<br>createEntry()→Entry,<br>hr(Entry,HR)→Entry,<br>bn(Entry,Cardinal)→Entry,<br>assign(Entry,Entry)→Entry,<br>compare(Entry,Entry)→Boolean,<br>compareHR(Entry,HR)→Cardinal,<br>compareInt(Entry,Interval)→Cardinal,<br>gHr(Entry)→HR,<br>dBn(Entry)→Cardinal,<br>destroyEntry(Entry)→. | -**createEntry()**: Constructor.<br>Create an empty entry<br>(emptyEntry).<br>-**hr()**: Mutator. Change the<br>value of the hyper-rectangle<br>of the entry.<br>-**bn()**: Mutator. Change the<br>value of the blob number of<br>the entry.<br>-**assign()**: Mutator. Change the<br>values of the first entry. |
| 2 | *Variables:*<br>Entry: e, emptyEntry<br>HR: h, emptyHR<br>Cardinal: bId | -**gHr()**: Observer. Return<br>the value of the hyper-rectangle<br>of the entry.<br>-**compare()**: Observer. Return |
| 3 | *Sample semantics:*<br>gHr(createEntry())=emptyHR<br>gBn(createEntry())=0<br>gHr(hr(e,h))=h<br>gBn(bn(e,bId))=bId<br>assign(createEntry(),e)=e<br>assign(e,createEntry())=emptyEntry | true if both entries<br>satisfy the comparison.<br>-**compareHR()**: Observer. Return<br>the blob number if both hyper-<br>rectangles satisfy the comparison.<br>-**compareInt()**: Observer. Return<br>the blob number if the interval in<br>dimension d satisfy the comparison<br>with the given interval.<br>-**gBn()**: Observer. Return the<br>blob number of the entry.<br>-**destroyEntry()**: Destructor. |

Figure 5.8: TDSO specification of the **Entry** class.

| Apr.96 | **Specification {16}Class RTreeNode** Classes: Boolean, Cardinal, Character, HR, Entry | |
|---|---|---|
| 1 | *Syntactic:*<br>createRTreeNode()→RTreeNode,<br>insert(RTreeNode,Entry,Cardinal)→RTreeNode,<br>delete(RTreeNode,Cardinal)→RTreeNode,<br>maxHb(RTreeNode)→HR,<br>assign(RTreeNode,RTreeNode)→RTreeNode,<br>readNode(RTreeNode,Cardinal)→RTreeNode,<br>writeNode(RTreeNode)→RTreeNode,<br>destroyRTreeNode(RTreeNode)→. | **-createRTreeNode()**: Constructor e initiator. Create an empty R_tree node (emptyRTreeNode). |
| 2 | *Variables:*<br>RTreeNode: rtn<br>HR: h<br>Entry: e ; Cardinal: p | **-insert()**: Mutator. Insert a new entry in the indicated position. |
| 3 | *Sample semantics:*<br>maxHb(createRTreeNode())=emptyHR<br>assign(createRTreeNode(),rtn)=rtn<br>assign(rtn,createRTreeNode())=emptyRTreeNode<br>readNode(rtn,0)=emptyRTreeNode<br>writeNode(createRTreeNode())=<br>emptyRTreeNode<br>delete(insert(createRTreeNode(),e,p),p)=<br>emptyRTreeNode | **-delete()**: Mutator. Delete the indicated entry.<br>**-maxHb()**: Observer. Return the hyper-rectangle of all the entries into the node.<br>**-assign()**: Mutator. Change the first node by the second one.<br>**-readNode()**: Mutator. Read from disk the node which is stored in the indicated blob number, if its blobId≠0.<br>**-writeNode()**: Mutator. Write the node on disk, if its blobId≠0.<br>**-destroyRTreeNode()**: Destructor. Write the R_tree node on disk before destroying it. |

Figure 5.9: TDSO specification of the **RTreeNode** class.

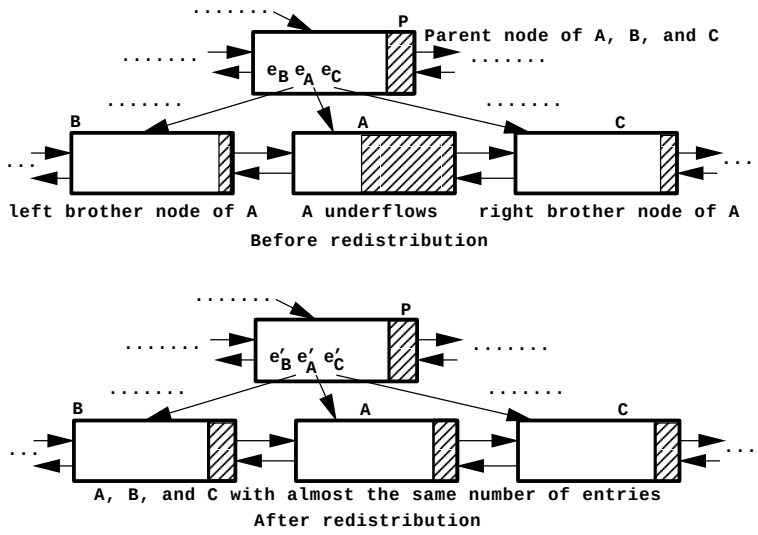



Figure 5.10: Three nodes redistribution of entries in case of deletion.

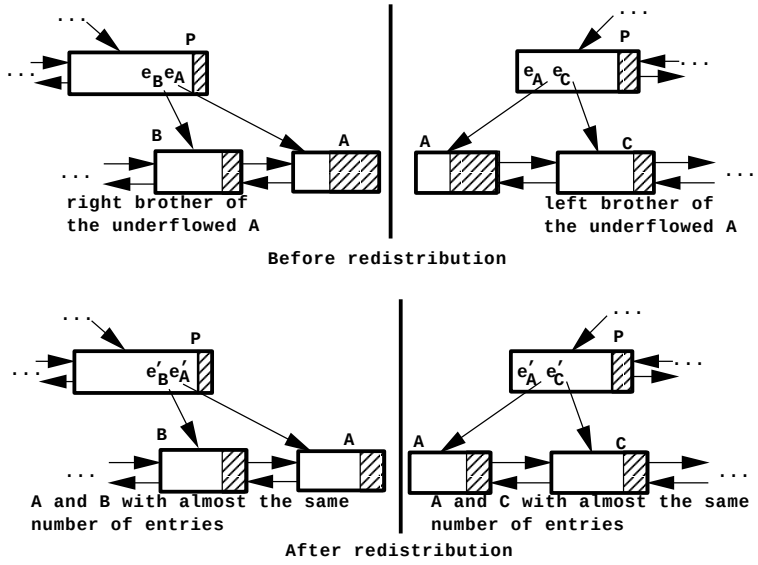



Figure 5.11: Two nodes redistribution of entries in case of deletion.

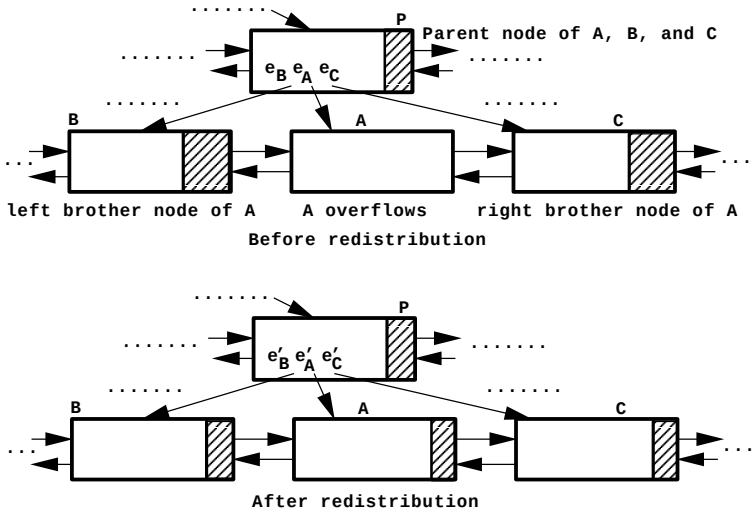



Figure 5.12: Three nodes redistribution of entries in case of insertion.

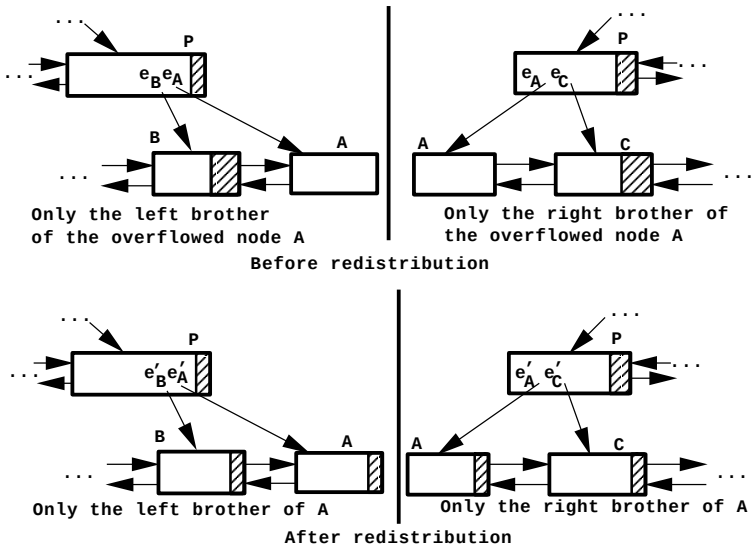



Figure 5.13: Two nodes redistribution of entries in case of insertion.

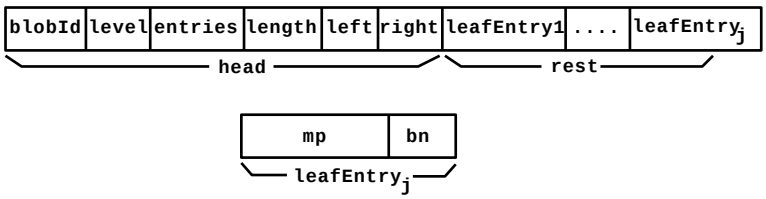



Figure 5.14: Leaf nodes format for point data.

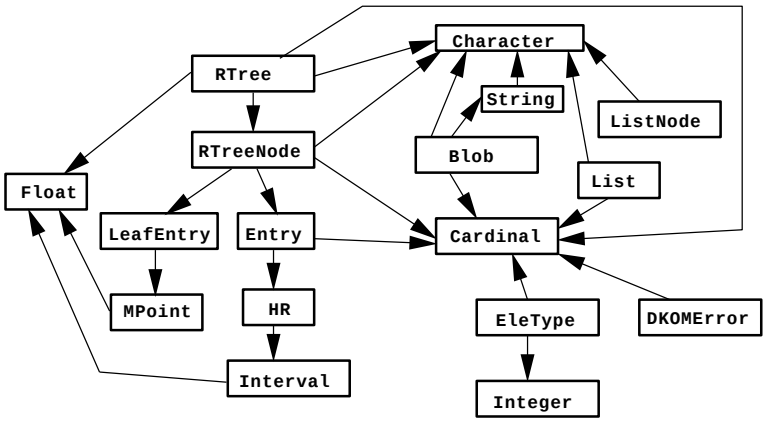



Figure 5.15: New TDSO diagram for the composition hierarchy.

| Apr.96 | **Specification {17}Class MRTree** Classes: Boolean, Entry, HR, Integer, ListOf | |
|---|---|---|
| 1 | *Syntactic:* | **-createMRTree()**: |
| | createMRTree()→MRTree, | Constructor and initiator. |
| | insert(MRTree,LeafEntry)→MRTree, | Create an empty R_tree. |
| | delete(MRTree,LeafEntry)→MRTree, | **-insert()**: Mutator. |
| | match(MRTree,MPoint)→Integer, | Insert a new leaf entry. |
| | search(MRTree,HR)→ListOf Integer, | **-delete()**: Mutator. |
| | cleanMRTree(MRTree)→MRTree, | Delete an existing |
| | isEmpty(MRTree)→Boolean, | leaf entry. |
| | destroyMRTree(MRTree)→. | **-match()**: Observer. |
| 2 | *Variables:* | Return the blob number |
| | MRTree: mrt, emptyMRTree | of this point. |
| | LeafEntry: e | **-search()**: Observer. |
| | HR: hr ; MPoint: mp | Return a list of blobId |
| 3 | *Sample semantics:* | corresponding to entries |
| | isEmpty(createMRTree())=True | that answer the search. |
| | isEmpty(insertEnt(createMRTree(),e))=False | **-cleanMRTree()**: |
| | isEmpty(deleteEnt(createMRTree(),e))=True | Mutator. Returns |
| | search(createMRTree(),hr)=emptyList | an empty R_tree. |
| | isEmpty(cleanMRTree(mrt))=True | **-isEmpty()**: Observer. |
| | search(cleanMRTree(mrt),hr)=emptyList | Return true if the R_tree |
| | match(cleanMRTree(mrt),mp)=0 | is empty, otherwise false. |
| | match(createMRTree(),mp)=0 | **-destroyMRTree()**: |
| | delete(insert(createMRTree(),e),e)=emptyMRTree | Destructor. Destroy |
| | | the R_tree. |

Figure 5.16: TDSO specification of the **MRTree** class.

| Apr.96 | **Specification {18}Class MPoint**<br>Classes: Float, Boolean, Cardinal | |
|---|---|---|
| 1 | *Syntactic:*<br>createMPoint()→MPoint,<br>change(MPoint,Float,Cardinal)→MPoint,<br>retrieve(MPoint,Cardinal)→Float,<br>compare(MPoint,MPoint)→Boolean,<br>assign(MPoint,MPoint)→MPoint,<br>destroyMPoint(MPoint)→. | **-createMPoint()**:<br>Constructor. Create an<br>empty dD point.<br>**-change()**: Mutator. The<br>float value in the specified<br>dimension is changed.<br>**-assign()**: Mutator. |
| 2 | *Variables:*<br>MPoint: i, emptyMPoint<br>Float: v<br>Cardinal: d | Multi-point assignment.<br>**-retrieve()**: Observer.<br>Return the float value<br>of the specified dimension. |
| 3 | *Sample semantics:*<br>retrieve(createMPoint(),d)=0<br>assign(createMPoint(),i)=i<br>assign(i,createMPoint())=emptyMPoint<br>retrieve(change(i,v,d),d)=v | **-compare()**: Observer.<br>Return true if both dD points<br>satisfy the comparison operator.<br>**-destroyMPoint()**: De-<br>structor. Destroy the interval. |

Figure 5.17: TDSO specification of the **MPoint** class.

| Apr.96 | **Specification {19}Class LeafEntry**<br>Classes: Boolean, Cardinal, MPoint | |
|---|---|---|
| 1 | *Syntactic:*<br>createLeafEntry()→LeafEntry,<br>mp(LeafEntry,MPoint)→LeafEntry,<br>bn(LeafEntry,Cardinal)→LeafEntry,<br>assign(LeafEntry,LeafEntry)→LeafEntry,<br>compare(LeafEntry,LeafEntry)→Boolean,<br>compareMp(LeafEntry,MPoint)→Cardinal,<br>gMp(LeafEntry)→MPoint,<br>gBn(LeafEntry)→Cardinal,<br>destroyLeafEntry(LeafEntry)→. | **-createLeafEntry()**:<br>Constructor. Create an<br>empty leaf entry.<br>**-mp()**: Mutator. Change<br>the mpoint value<br>of the leaf entry.<br>**-bn()**: Mutator. Change<br>the blob number<br>of the leaf entry.<br>**-assign()**: Mutator.<br>Leaf entry assignment. |
| 2 | *Variables:*<br>LeafEntry: le, emptyLeafEntry<br>MPoint: m, emptyMPoint<br>Cardinal: bId | **-compare()**: Observer.<br>Return true if both entries<br>satisfy the comparison. |
| 3 | *Sample semantics:*<br>gMp(createLeafEntry())=emptyMPoint<br>gBn(createLeafEntry())=0<br>gMp(mp(le,m))=m<br>gBn(bn(le,bId))=bId<br>assign(createLeafEntry(),le)=le<br>assign(le,createLeafEntry())=emptyLeafEntry | **-compareMp()**: Observer.<br>Return the blob number if<br>both multi-point values<br>satisfy the comparison.<br>**-gMp()**: Observer. Return<br>the multi-point value.<br>**-gBn()**: Observer.<br>Return the blob number.<br>**-destroyLeafEntry()**:<br>Destructor. |

Figure 5.18: TDSO specification of the **LeafEntry** class.

| Apr.96 | **Specification {20}Class RTreeNode** Classes: Boolean, Cardinal, Character, MPoint, HR, LeafEntry, Entry | |
|---|---|---|
| 1 | *Syntactic:*<br>createRTreeNode()→RTreeNode,<br>insLeaf(RTreeNode,LeafEntry,Cardinal)→<br>RTreeNode,<br>insert(RTreeNode,Entry,Cardinal)→RTreeNode,<br>delete(RTreeNode,Cardinal)→RTreeNode,<br>maxHb(RTreeNode)→HR,<br>assign(RTreeNode,RTreeNode)→RTreeNode,<br>readNode(RTreeNode,Cardinal)→RTreeNode,<br>writeNode(RTreeNode)→RTreeNode,<br>destroyRTreeNode(RTreeNode)→. | **-createRTreeNode()**:<br>Constructor. Create<br>an empty node.<br>**-insLeaf()**:<br>Mutator.<br>Insert a new<br>leaf entry in the<br>indicated position.<br>**-insert()**: Mutator.<br>Insert a new entry in<br>the given position. |
| 2 | *Variables:*<br>RTreeNode: rtn, emptyRTreeNode<br>HR: h, emptyHR<br>Entry: e | **-delete()**: Mutator.<br>Delete the<br>indicated entry.<br>**-maxHb()**: Observer. |
| 3 | *Sample semantics:*<br>maxHb(createRTreeNode())=emptyHR<br>assign(createRTreeNode(),rtn)=rtn<br>assign(rtn,createRTreeNode())=emptyRTreeNode<br>writeNode(createRTreeNode())=emptyRTreeNode<br>readNode(rtn,0)=emptyRTreeNode | Return the *mbhr*<br>of the RTreeNode.<br>**-assign()**: Mutator.<br>RTreeNode assignment.<br>**-readNode()**: Mutator.<br>Read from the node<br>which is stored in the<br>indicated blob number,<br>if its blobId≠0<br>**-writeNode()**:<br>Mutator. Write the<br>node on disk.<br>**-destroyRTreeNode()**:<br>Destructor. Write<br>the node on disk<br>before destroying it. |

Figure 5.19: New TDSO specification of the **RTreeNode** class.

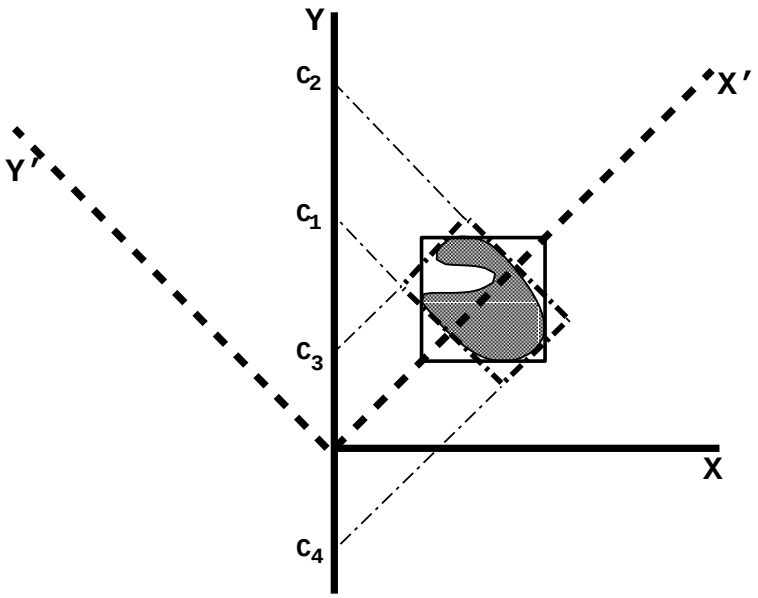



Figure 5.20: Two *mbr*s over a polygonal object.

# Chapter 6

# Test of the R_tree extensions

In order to compare the R_tree extensions, experiments based on four different kinds of data were designed. The first dataset comprised randomly generated *mbr*s in 2D, uniformly distributed in space. For the second dataset, we used real data from the Radioactivity Environmental Monitoring (REM) database containing spatio-temporal points. The third dataset comprised polygonal real data containing the counties of the UK and the wards of the West Yorkshire county. For the last data set, spatio-temporal data uniformly distributed in space and time were generated as *mbhr*s in 3D, but for lack of time, we only used this three times, once to test extension 1, second to test extension 3, and the other to test extension 4.

The experiments were limited in a number of range, partly due to lack of real datasets of sufficient size. Future work will extend the experiments by looking at clustering datasets and also spatio-temporal polygonal data. Most of the conclusions in this chapter are based on simulated spatial data.

This chapter describes the main characteristics of the data used, how they were generated (where appropriate), what sort of data they contain, how the experiments were carried out, what kind of transformations were made to obtain the required data, and the analysis of the results obtained.

## 6.1 Spatial data

This section describes the type of data that we used to test the R_tree extensions in 2D. The spatial data used was expressed either by the two coordinates of a *mbr* or by the four coordinates of a *tbo*. The *mbr*s were utilised for the extensions 1 to 4, and the *tbo*s for the O_tree.

### 6.1.1 Data uniformly distributed in space

The first step for testing the extensions of the R_tree structure required *mbr*s in 2D. Because the data of the Carora city (mentioned in chapter 2) was not available to us at that time, we decided to generate a file containing *mbr*s for objects uniformly distributed in space with their height and width also uniformly distributed. We followed similar ideas as used in simulation to generate data, by using a uniform function that first generates a pseudo-random number uniformly distributed over the interval [0, 1) and later transforms it into a numberthat follows a uniform distribution over a different interval. In this case, we use a uniform distribution between 0 and 800 for generating the centroid of the *mbr*, a uniform distribution between 1 and 100 to generate the width of the rectangle, and finally, a uniform distribution between 10 and 100 to generate the height of the rectangle.

Figure 6.1 illustrates the TDSO form of the implementation of this program, called **MBRCreation**, and the uniform function that it uses. Each time the program is executed, it generates the same data because we do not use different seed values and the uniform function uses the same pseudo-random number sequence generated by one of the available C++ library functions named *drand48()*. The C++ library function returns a non- negative double-precision floating point value uniformly distributed over the interval [0.0, 1.0) using the linear congruential algorithm and 48-bit integer arithmetic. The linear congruential formula gives $X_{n+1} = (wX_n + c)\ mod\ m\ with\ n \geq 0$, where $m = 2^{48}$, $w = (273673163155)_8$, $and\ c = (13)_8$. The value returned by drand48() is computed by first generating the next 48-bit X in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (left most) bits of Xi and transformed into the returned value. The program can be used for generating the required number of *mbr*s uniformly distributed in the space described above. An example of the data obtained is also presented in this figure.

| May.96 **Implementation MBRCreation** Classes: Float, Integer, Entry | | |
|---|---|---|
| 1<br>2<br>3 | nrec=Read from the keyboard<br>mbrFile=File<br>[ xc=uniform(0,800)<br>yc=uniform(0,800)<br>xa=uniform(1,100)<br>yl=uniform(10,100)<br>mbrRec=(xc-xa/2, yc-yl/2, xc+xa/2,<br>yc+yl/2,i+1)<br>Write mbrRec in mbrFile<br>] i=0,nrec | **-nrec**: Integer: Number of *mbr* that will be generated.<br>**-mbrFile**: String: File where are stored the generated data.<br>**-xc,yc**: Float: Centroid of a *mbr*.<br>**-xa,yl**: Float: Width and large of a *mbr*.<br>**-mbrRec**: Entry: *mbr* and number of the record in the file. |
| 1 | nrec=2, mbrFile='s2D.mbr'<br>mbrRec=(299.182 335.162 647.292 697.485 1)<br>mbrRec=(253.683 256.226 677.859 740.427 2) | Create two *mbr*s following the defined distribution. |

| May.96 **Implementation uniform(Float: a, b): Float** Classes: Float | | |
|---|---|---|
| 1 | return (a+(b-a)*drand48()) | **-drand48()**: C++ library function that generates pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic. |

Figure 6.1: TDSO implementation of the **MBRCreation** program.

### 6.1.2 Polygonal data

In order to test the efficacy of the O_tree described in the last chapter, some files from Manchester Computing Center were obtained: one containing the segments of the wards of the West Yorkshire county of the U.K. and the other containing the segments of the counties of the U.K.

These real data were stored following the format shown in figure 6.2. Each record file contains the information about a segment, beginning with the names of two polygons, to identify the polygon on the left, and the polygon on the right of this segment. These names are codes defining each division of either a county or the country. After that appears the first point of the segment given as an absolute value, and followed by a set of points expressed in relative coordinates, where each is associated with its predecessor. To calculate a new absolute coordinate based on its

predecessor, the current absolute coordinate is added to the next relative coordinate value. In the example shown below, the second absolute coordinate is 39956 41295. An example of these records is the following, Z03BPAP Z08CZAU 39951 41301 6 -6 10 -13 6 -8 5 -5 ... 5 -7 2 -3 2 -4 /. Polygons pertaining to the West Yorkshire county began with Z08, and the corresponding ascii file is shown in appendix C.

| Left segment identifier | Right segment identifier | Absolute coordinates X and Y | Set of relative coordinates X and Y in relation to its predecessor. |
|---|---|---|---|

Figure 6.2: Format of each segment of polygonal data files.

These files were processed by a program called *tbo* that calculates the tight bounding octagon (*tbo*) of each polygon in the file. A *tbo* contains the two coordinates used for the two *mbr*s, as mentioned in chapter 4. The corresponding TDSO implementation of the **tbo** program is presented in figure 6.3. The identification of each polygon is made by its code (name), and it can appear several times in the file. Files can have polygon names of other counties which are its neighbours, those neighbour polygons are not included in the output files.

## 6.2 Spatio-temporal data

The term spatio-temporal data is used for data that includes time values related to the 2D *mbr*. The implemented R_tree extensions three, four, and five support many dimensions in space and time, however we have not tested it with more than two dimensions in geographic space and one dimension in time.

### 6.2.1 Interval data uniformly distributed in space and time

Due to the lack of real spatio-temporal data expressed as intervals, we decided to generate a set by using the **MBHRCreation** program that generates the file named *st3D.mbhr*. This program is basically the **MBRCreation** program shown in the last section to generate spatial data. The spatio-temporal intervals generated by the program are uniformly distributed in space and time. It uses the same function and distributions for generating *mbr*s, and generates time intervals between 01/01/1990 and 31/12/1995. The extent of the time intervals is also uniformly distributed in [0,364]. Figure 6.4 presents the TDSO implementation of the program and one example of the generated minimum bounding hyper-rectangles (*mbhr*s).

### 6.2.2 Point data

The REM databank contains environmental monitoring data collected after the nuclear accident of Chernobyl city in the USSR. These data correspond to several readings of ground-deposited CS-137 at various spatial and temporal locations. Roberts presents in [Rob95] a preliminary analysis of the data that highlights its main characteristics.

Firstly, the data is not uniformly distributed either in space or in time. This non-uniformity is because some monitors are at fixed locations and provide readings regularly spaced in time, and others are mobile and take readings at those locations considered to be most significant data at that time. Second, data values for location are expressed in absolute coordinates as latitude and longitude. Third, some readings have a date where the data values were taken, and these values are expressed in absolute time as day, month, and year. Finally, location-time data values correspond to spatio-temporal points.

For this application, the spatio-temporal access method will take location and time values as real numbers having a granularity of meters and microseconds, respectively. The problem of different scales is left to the spatio-temporal query language, as is the problem of having only relative values. The index only supports absolute values.

The data provided by this application is used to test the R_tree extension 5 that was defined in the last chapter. This data has the format presented in figure 6.5, a code determining the administrative district for Europe (NUTS code), the name of the district (locality), the code of the sample type that describes how the sample was obtained, a reference to the source of the data, the date of the sample on which the measurement was taken, the nuclide whose concentration is being measured (nuclide), the measurement value, the units of measurement, degrees latitude, and degrees longitude. An example of the records is the following, 11, SCHLESWIG-HOLSTEIN, A112D, DWDO, 29-apr-1986, I-131, 0.000, BQ/M3, 54.434, 9.514.

## 6.3 Experimental design

Each of the first four extensions were prepared for execution on the same platform varying the same experimental parameters, that is: blob size (512, 1024, and 2048) and number of objects (1000, 2770, 5000, and 10000). We choose 2770 because this is the maximum file size for the REM databank available for evaluating extension

5. The multi-dimensional point R_tree (extension 5) was tested with the same blob sizes, and the number of objects in (1000, 2770), due to the quantity of available data. The last extension, the O_tree, was only tested with the polygonal data files which were processed by the *tbo* program creating the **.tbo** testing files. All of the extensions were implemented on C++ on a UNIX platform, and tested on the same standalone Silicon Graphics machine.

### 6.3.1 Parameters

The comparison was based on the parameters which measure the main characteristics of a tree, called the measured parameters. These measured parameters are presented in figure 6.6.

These parameters will be used to describe and compare the extensions in the section of the results.

### 6.3.2 Test programs

The test programs differ in the number of the implemented operators depending on the extension to be tested. Thus, the **e1** program was used to test the extension 1, **e2** to test extension 2, and so on. Figure 6.7 shows the general TDSO implementation corresponding to all the test programs. The rest of the TDSO implementations for functions called by a test program are presented in appendix C.

Insertion and searching methods into the **RTree** class include some statements to measure time. We use the timing function provided by C++ in UNIX systems that is *getrusage()*. This function returns information describing the resources utilized by the current process, or all its terminated child processes. We use getrusage(RUSAGE_SELF, time) for getting initial and final time of each operations that we need to measure. For each operation e.g. insert, retrieve, etc., we transform the time obtained to seconds, and compute average and variance over all objects in the file. These two calculated values were utilized to perform some statistical tests mentioned in the next section. To be sure that time values were as accurate as possible, we put initial time measure statement inside the function just before the beginning of the real work of it, and the final time measure statement just after the real work finished inside the function. After that instruction, we put other statements to calculate time in seconds, time cumulation, and other things. We also avoid the timing of initialization of variables.

Each test program creates the structure by inserting one by one the object's mbr

and its reference. Each insertion gives an insertion time in seconds and the number of disk accesses. After inserting the demanded number of objects (1000, 2770, 5000, and 10000), the parameters H, NB, NE, ES, AIT, IDA, and the variance of insertion time are calculated.

For retrieval operations in all of the extensions, the test scenario is as follows:

1. AMST, MDA, MRO, and the variance of exact match time are calculated for exact match operations by searching for each object in turn.

2. ARST, RDA, RRO, and the corresponding variance are calculated for range operations. For every object in the dataset, the range operator is applied to find all the objects satisfying the range condition. This can be seen in tables of the appendix C.

3. For the rest of the retrieval operations, the same considerations apply.

## 6.4 Results

First of all, extension 1 is considered equivalent to the original R_tree because of the only difference between them is the deletion algorithm, and we do not use that method in any of our experiments. For this reason, in what follows, the results of extension 1 will be taken to represent the original R_tree and provide the reference point for evaluating the other extensions. Extensions 1 and 2 were tested with the 2D spatial data uniformly distributed in space, that is the *s2D.mbr* file generated by *MBRCreation* program. Figure 6.8 presents the curves obtained for empty space within nodes (ES) vs. database size. This figure shows clearly the reduction in wasted space within nodes for the case of extension 2. The reason for that is the use of the redistribution policy during insertions.

Referring to the average insertion time (AIT), figure 6.9 illustrates that there is no appreciable difference among the results obtained for both extensions. Extension 2 gives better AIT when the database size is larger than 10.000 of objects. On the other hand, there is an important difference in the results for the average exact match time (AMST) for the two extensions as shown in figure 6.10. It is clear that extension 2 always gives a greater AMST value than extension 1. This is because of the redistribution of entries during insertions. Surprisingly, the curve for the average range search time (ARST) shown in figure 6.11 illustrates no differences at all among the two extensions, that is redistribution during insertion does not affect

range retrieval because the function utilised to search into the nonleaf nodes is the same in both extensions (non disjoint).

Figure 6.12 presents the curves for the hit ratio vs. database size among the two extensions. This hit ratio was calculated as the number of retrieved objects divided by the number of disk accesses, to account for false hits. Thus, for exact match searches, the hit ratio was calculated as MRO/MDA, and similarly for range searches where it is defined as RRO/RDA. A greater hit ratio means less disk accesses are needed to retrieve the required objects. It follows that extension 1 requires fewer disk accesses than extension 2 for retrieving the same number of objects.

The complete results obtained for several runs are shown tabularly in appendix C. These runs correspond to the parameters mentioned in section 6.3.

Extensions 3 and 4 were tested with the same 2D spatial data used to test extensions 1 and 2. Figure 6.13 shows the reduction in waste space within nodes for the case of extension 4 for all database sizes. The reason for this is that extension 4 was built based on extension 2. Figure 6.14 presents lower average insertion times for extension 3, except for a database size of 5000, where they are equal. It means that redistribution in insertions have a cost in tree performances, when an insertion operation is made. For retrieval times, figure 6.15 clearly shows lower average exact match times for extension 3, because the sequential search within nodes is shorter than those of extension 4. Similar results are shown in figure 6.16 corresponding to a particular case of range retrieval, such as average *not disjoint* time (A!NT). In this particular point, we can observe a better time discrimination in extension 3 and 4, compared with the obtained results of extensions 1 and 2, where we obtain no differences at all in range searches.

The curve of average *disjoint* time (ADT) is not included here, because it is not very important in our comparison. A disjoint search means a complete scroll of tree leaf nodes and it normally retrieves all of the objects in the database, except the object of the search.

The calculated hit ratios for each case of extensions 3 and 4 are presented in the next figures. The hit ratio for exact *match* retrieval, MRO/MDA, is illustrated in figure 6.17, where hit ratio for extension 4 is clearly lower than those results of extension 3. Similar results were obtained for *not disjoint* (!DRO/!DDA), *inside* or *contain* (ICRO/ICDA), and *not disjoint in one dimension* (!D1RO/!D1DA) hit ratio curves. Conversely, the hit ratio for *disjoint* retrieval (DRO/DDA), *left* (L1RO/L1DA) and *right* (R1RO/R1DA) *in one dimension* hit ratios, are lower for extension 3 than for extension 4, except for a database size of 5,000. The redistri-

bution in insertion affects hit ratios in *disjoint*, and *left* and *right in one dimension* retrievals. The *cover* or *covered by* hit ratio is not present because the nature of the data is such that there will always be zero objects for CRO. The rest of the figures for each hit ratio type is presented in appendix C, and also four tables containing results of each program test.

For different combinations of the results, we made statistical tests to reinforce our conclusions. An analysis of two factors applied to the first four extensions vs. the use of different blob sizes gives a non significative difference with a confidence level of 95%, that is, blob size does not influence the extensions' behaviour. The comparison of means between AMST of extensions 2 and 4, for all of the blob size and number of objects, concludes that the use of the *covers* operator in extension 4 instead of the *overlap* of extension 2, decreases appreciably the AMST in successful searches. Figure 6.15 illustrates AMST for the first four extensions. Using the same type of test with extensions 3 and 4, we obtain a better AMST in extension 3. This is due to the lack of redistribution during insertions, even if it uses the *covers* operator. Example of this test:

H0: Initial hypothesis. AMST of both extensions 4 and 3 are equal. ($\mu_1 = \mu_2$).

H1: Alternative hypothesis. AMST of extension 4 is greater than AMST of extension 3. ($\mu_1 > \mu_2$).

where $\mu_1$: population mean for exact match search time of extension 4, and $\mu_2$: population mean for exact match search time of extension 3. For a confidence level of 99.8% and a blob size of 2K, we have

| Extension | Number of exact match searches in each case | Calculated mean | Variance |
|---|---|---|---|
| 3 | $n_1$=10000 | $\overline{X}_1 = 0.130$ sec | $S_1^2 = 0.113$ sec |
| 4 | $n_2$=10000 | $\overline{X}_2 = 0.072$ sec | $S_2^2 = 0.664$ sec |

and we obtain the estimated Z = 6.614, which is greather that $Z_{0.998} = 2.88$ and H0 is rejected. That means there is a significant difference at 99.8% between extension 3 and 4 due to the use of redistribution in insertion. AMST is better in extension 3.

On the other hand, a better AIT value was obtained by using *covers* operator without redistribution in insertions (extension 3). AIT curves for extensions 1, 2, 3, and 4 are shown in figure 6.14. It can also be seen the exact match hit ratios for these extensions in figure 6.17 where extension 3 is clearly better that the others. Figure 6.18 presents the range hit ratios for these four extensions, extensions 1 and

2 have this parameter, but it is not the case for extensions 3 and 4. In this particular case, we consider the **not disjoint hit ratio** of these two extensions as equivalent to the **range hit ratio** of extensions 1 and 2, because the *not disjoint* operator is used to calculate overlapping mbrs in extensions 3 and 4, and *range* is used to calculate the same in the other two extensions.

Two tests were made for extensions 3 and 4 using spatio-temporal data uniformly distributed in space and time, but only in the case of 10.000 of objects. For lack of time, we did not perform more tests, but in both cases the relationships between all measured results of extension 3 and 4 were maintained as for spatial tests. Results are included in appendix C in tabular form.

The multipoint version of the R_tree, that is extension 5, was tested with the REM data, but it could not be compared with other results because the lack of similar data. Figure 6.19 illustrates the average insertion time obtained for 1.000 and 2.770 of records of the preprocessed real data of REM.

Extension 6 is not comparable with any of the former extensions because of the different conception and data that it manages. It was tested with real polygonal data of the U.K. Unfortunately, after processing these two available data files, only a small number of records were obtained, 128 and 68 records, in the wards of West Yorkshire and the counties of the U.K., respectively. This quantity of records is so small that we decide to discard figures of this extension. Appendix C contains some obtained results for the wards of West Yorkshire.

## 6.5 Concluding remarks

A substantial improvement in terms of reduction in wasted space through the redistribution policy is our first conclusion. This policy mainly affects the average exact match search time and the search hit ratio, but it permits to obtain a more compact index structure.

Secondly, we have obtained a better hit ratio and therefore fewer disk accesses by including a minimal set of more specific and specialized search operators that were defined in chapter 4.

Third, we have shown that the object-oriented approach and the TDSO technique lead to a very flexible scheme whereby the indexing structure can be very easily adapted to the problem at hand. As an example of that, we had developed quickly and easily the O_tree extension by only adding a few new methods to the extension 3 class. This point is important because we are trying to accommodate requirements

that appear at the query stage right down at the indexing level. Consequently, we can say that whatever requirements an application presents, the object-oriented code can easily be adapted to meet them.

The benefits of the comparison is to know which parameter values are appropriate for better R_tree performance. The outcome of the comparison is basically that AIT is better in extension 3 due mainly to the use of the *covers* operator inside the majority of the internal range search routines of the **Rtree** class. AMST is also better in extension 3. There is no appreciable difference in using a particular blob size for any extension used, so the size of the blob is not an important performance factor as we had thought at the beginning of this work. Height (H), number of entries (NE), number of used blobs (NB), and empty space (ES) are reduced in those extensions with redistribution during insertion, i.e., extensions 2 and 4. The cost of having a compacted R_tree index with better AIT is paid for in terms of the time taken for retrieving a specific entry in the structure. In general, extension 3 offers the best performances, except for *disjoint*, and *left and right in one dimension* hit ratios. This is due to the overlapping regions in the higher nodes of the tree, where the left and right in one dimension routines have to examine almost all of the tree nodes.

Comparing these four extensions together for the same data file (s2D.mbr), we have obtained the following results:

- in terms of the reduction on wasted space within nodes, extension 4 is the best for all database sizes,
- in terms of average insertion time, average exact match time, and exact match hit ratio, extension 3 is the best,
- if it is valid to compare range searches of extensions 1 and 2 with *not disjoint* searches, then we can say that
  - in terms of average range time, extensions 1 and 2 are the best, and
  - the range hit ratio was better in extension 3.

Extensions 1 and 2 can not be compared directly with extensions 3 and 4 to have some curves of the main parameters. Nevertheless, extensions 1 or 2 can be used to retrieve a set of objects with a poor hit ratio for those queries different to exact match or *not disjoint* queries. For example if the question is *Find all objects that cover object A* and the standard R_tree index of extension 1 is used, then it is

retrieved all objects that are *not disjoint* from A and then selected from those just the ones that cover A. Whilst that the same query can be answered more quickly with extension 3 or 4, because instead of using the *not disjoint* operator, it is used the *cover* operator which retrieve a smaller set of retrieved objects in the first phase to answer that query.

Extension 5 and 6, the multipoint R_tree and the O_tree, have acceptable performances, mainly because we used extension 3 as a base to build them. Extension 5 has entries of smaller length in the leaf nodes than those of the O_tree and even the other extensions. Extension 6 has the leaf node entries of largest length. Extension 5 has the most compact index, and extension 6 is the largest index tree.

The lack of appropriate real data to test the R_tree extensions is a point that we can conclude. The generation of adequate data to each extension was made easily without major complications. The parameters for this generation were chosen randomly. The use of the uniform distribution was selected as a first case of testing, having no available time to generate data following other distributions.

| Jun.96 **Implementation tbo** Classes: Integer, String, SetOf, Polygon, File | | |
|---|---|---|
| 2<br><br><br><br><br><br><br><br><br><br><br><br><br><br><br>3<br>2 | for each line in the file of segments<br>[ endOrPolyId(line,pn1,pn2,pos1,<br>pos2,xa,ya)<br>searchPolyName(pn1,cnp,pol)<br>searchPolyName(pn2,cnp,pol)<br>if(pos1) [ pos1=insNew(pn1,cnp,pol)]<br>if(pos2) [ pos2=insNew(pn2,cnp,pol)]<br>if(xa and ya) [ initiates pol(pos1).tbo<br>and pol(pos2).tbo]<br>for each relative coordinate<br>[ update xa and ya<br>update pol(pos1).tbo and<br>pol(pos2).tbo]<br>]<br>nrf=writePoly(pol,ifTbo,ifTxt)<br>Define ifTbo and ifTxt | **-line**: String: Record of the file of segments.<br>**-pn1,pn2**: String: Names of the polygons one and two.<br>**-pos1,pos2**: Integer: Position of each polygon in the set of polygons.<br>**-xa,ya**: Integer: Actual values of the absolute coordinates.<br>**-cnp**: Integer: Polygon counter.<br>**-pol**: SetOf Polygon: Contain name and *tbo* of each polygon.<br>**-ifTbo,ifTxt**: File: Binary and text file containing the set of polygons.<br>**-nrf**: Integer: Polygon number in this region.<br>**-endOrPolyId(),writePoly(), searchPolyName(),insNew()**: Functions defining new polygons, searching polygons in the set, inserting a new polygon in the set, and writing the set of polygons in two files. |
| 1 | 38820.00 45282.00 40283.00 45403.00<br>81074.00 89464.00 -3974.00 4900.00<br>3 Z08 | Record in the ifTxt file containing the *tbo* of the West Yorkshire county. |

Figure 6.3: TDSO implementation of the **tbo** program.

| May.96 **Implementation MBHRCreation** Classes: Float, Integer, String, Entry | | |
|---|---|---|
| 1<br>2<br>3 | nrec=Read from the keyboard<br>mbrFile=File<br>[ xc=uniform(0,800)<br>yc=uniform(0,800)<br>xa=uniform(1,100)<br>yl=uniform(10,100)<br>y=uniform(1,6)<br>mo=uniform(1,12)<br>depending on y and mo di=uniform(1,31)<br>ti=y/mo/di<br>ed=uniform(0,364)<br>depending on mo ed=f1(ed,nd,di,mo,y)<br>ts=y/mo/di<br>mbhrRec=(xc-xa/2, yc-yl/2,xc+xa/2,<br>yc+yl/2, ti, ts, i+1)<br>Write mbhrRec in mbrFile<br>] i=0,nrec | **-nrec**: Integer: Number of *mbhr* that will be generated.<br>**-mbrFile**: String: File where are stored the generated data.<br>**-xc,yc**: Float: Centroid of a *mbr*.<br>**-xa,yl**: Float: Width and large of a *mbr*.<br>**-y,mo,di**: Float: Year, month, and day, respectively.<br>**-ti,ts**: Float: Low time and high time of the interval.<br>**-ed**: Float: Extent of the time interval in days.<br>**-mbhrRec**: Entry: *mbhr* and number of the record in the file. |
| 1 | nrec=2, mbrFile='st3D.mbhr'<br>mbr:(315.9 318.443 641.104 703.672<br>19910510 19910704 1)<br>mbr:(119.121 135.869 277.985 335.961<br>19930128 19930204 2) | Create two *mbhr*s following the defined distribution. |

Figure 6.4: TDSO implementation of the **MBHRCreation** program.

| NUTS code | Name of the district | Sample Type | Reference | Date of the sample | Nuclide | Measure ment value | Units of measure ment | Latitude | Longitude |
|---|---|---|---|---|---|---|---|---|---|

Figure 6.5: Format of each record of the REM data file.

| *Parameter* | *Meaning* |
|---|---|
| H | The height of the tree, which measures, in theory, how many disk accesses are needed to find an entry in the tree. |
| NB | The total number of used blobs, that express the actual disk space used by the tree. |
| NE | The total number of entries, equivalent to the tree size. |
| ES | The percentage of empty space within blobs, which measures the space occupancy of each blob. |
| AIT | The average insertion time, that shows how fast the structure is during an insertion. |
| AMST | The average *exact match* search time, that shows how fast an exact match search is in the current tree. |
| ARST | The average *range* search time, which gives the average timing of a range search in the actual tree (extensions 1 and 2). |
| ADT | The average *disjoint* search time, that shows how fast a disjoint search is in the actual tree (extensions 3 and 4). |
| A!DT | The average non disjoint search time, which gives the average timing of a *meet* or *overlap* operation (extensions 3 and 4). |
| AICT | The average *inside* or *contain* search time, that shows the average timing of a containment search (extensions 3 and 4). |
| ACT | The average *cover* or *coveredBy* search time (extensions 3 and 4). |
| A!D1T | The average *non disjoint in one dimension* search time, which shows the average timing of a non disjoint in a given dimension search (extensions 3 and 4). |
| AL1T | The average *left in one dimension* search time (extensions 3 and 4). |
| AR1T | The average *right in one dimension* search time (extensions 3 and 4). |
| IDA | The average number of disk accesses during an insertion, saying how many disk accesses in average were made to insert a new entry in the tree. |
| MDA | The average number of disk accesses during an exact match search, that express how many disk accesses in average are required to search for an entry in the tree. |
| RDA | The average number of disk accesses during a range search. This measures how many disk accesses in average are required to search all the tree entries which satisfy the condition of a specified range operator for a given region or *mbr* (extensions 1 and 2). |
| DDA | The average number of disk accesses during a *disjoint* search (extensions 3 and 4). |
| !DDA | The average number of disk accesses during a *non disjoint* search (extensions 3 and 4). |
| ICDA | The average number of disk accesses during an *inside* or *contain* search (extensions 3 and 4). |
| CDA | The average number of disk accesses during a *cover* or *coveredBy* search (extensions 3 and 4). |
| !D1DA | The average number of disk accesses during a *non disjoint in one dimension* search (extensions 3 and 4). |
| L1DA | The average number of disk accesses during a *left in one dimension* search (extensions 3 and 4). |
| R1DA | The average number of disk accesses during a *right in one dimension* search (extensions 3 and 4). |
| MRO | The average number of retrieved objects during an exact match search. This number must be equal to 1 in all cases. |
| RRO | The average number of retrieved objects during a range search (extensions 1 and 2). |
| DRO | The average number of retrieved objects during a *disjoint* search, which is always equal to the total number of objects less one (extensions 3 and 4). |
| !DRO | The average number of retrieved objects during a *non disjoint* search, that is for *meet* or *overlap*(extensions 3 and 4). |
| ICRO | The average number of retrieved objects during an *inside* or *contain* search (extensions 3 and 4). |
| CRO | The average number of retrieved objects during a *cover* or *coveredBy* search (extensions 3 and 4). |
| !D1RO | The average number of retrieved objects during a *non disjoint in one dimension* search (extensions 3 and 4). |
| L1RO | The average number of retrieved objects during a *left in one dimension* search (extensions 3 and 4). |
| R1RO | The average number of retrieved objects during a *right in one dimension* search (extensions 3 and 4). |

Figure 6.6: The measured parameters.

| Jun.96 **Implementation testProgram** Classes: Integer, String | | |
|---|---|---|
| 1<br>2<br>3 | argc=Read from the keyboard<br>argv=Read from the keyboard<br>if(argc=5)<br>[ file=Read from the keyboard<br>recl=Read from the keyboard<br>fs=Read from the keyboard<br>op=Read from the keyboard<br>ent=Read from the keyboard<br>from=Read from the keyboard<br>top=Read from the keyboard<br>dop=Read from the keyboard<br>prox=Read from the keyboard<br>tep=Read from the keyboard<br>hr1=Read from the keyboard<br>hr2=Read from the keyboard<br>]<br>if(argc=13 ∨ argc=26)<br>[ from=ent=op=top=dop=prox=<br>tep=file=recl=fs=hr1=hr2=Substring of argv<br>]<br>choices(from, ent, op, top, prox, dop, tep, recl, file, hr1,hr2) | **-argc**: Integer: Number of calling parameters.<br>**-argv**: String: Calling parameters.<br>**-file**: String: Data file name.<br>**-recl**: Integer: Number of bytes of each file record.<br>**-fs**: Integer: Number of file records.<br>**-op**: Integer: Operation number.<br>**-ent**: Integer: Number of records to be tested.<br>**-from**: Integer: Record number from which begins the records to be tested.<br>**-top, prox, dop, tep**: Number of the topological, proximity, directional, and temporal operators, respectively.<br>**-choices()**: Function defining the menu, respectively. |
| 1<br>3<br>4 | argc=2<br>argc=5<br>argc=13 | Invalid argument<br>Using the test menu<br>Simulation run |

Figure 6.7: TDSO implementation of a **testProgram** program.

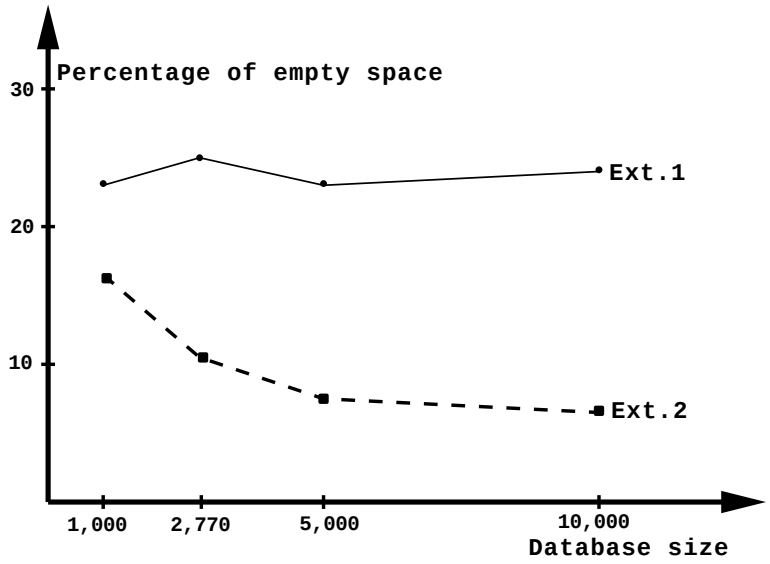



Figure 6.8: Percentage of empty space vs. data set size for extensions 1 and 2. (File: s2D.mbr, blob size=2KB)

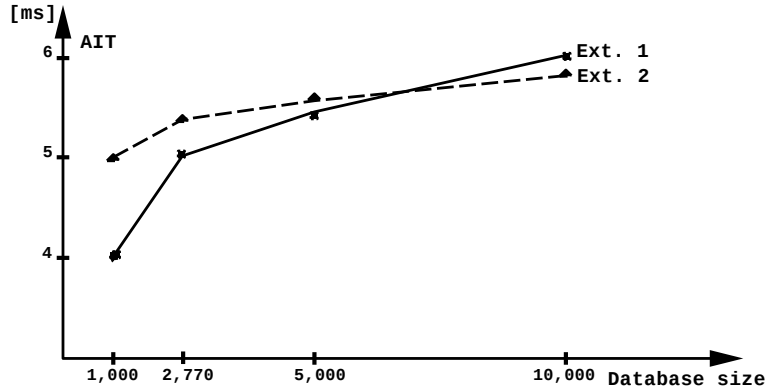



Figure 6.9: AIT vs. data set size for extensions 1 and 2. (File: s2D.mbr, blob size=2KB)

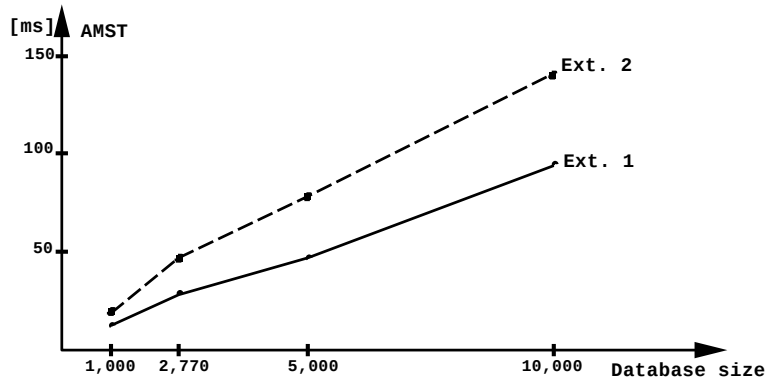



Figure 6.10: AMST vs. data set size for extensions 1 and 2. (File: s2D.mbr, blob size=2KB)

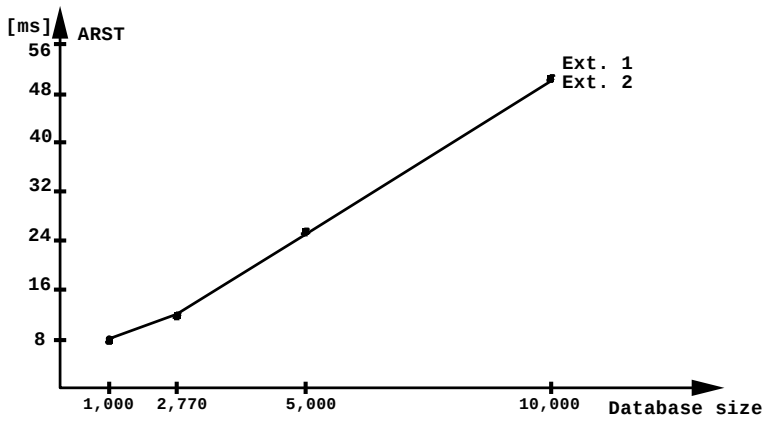



Figure 6.11: ARST vs. data set size for extensions 1 and 2. (File: s2D.mbr, blob size=2KB)

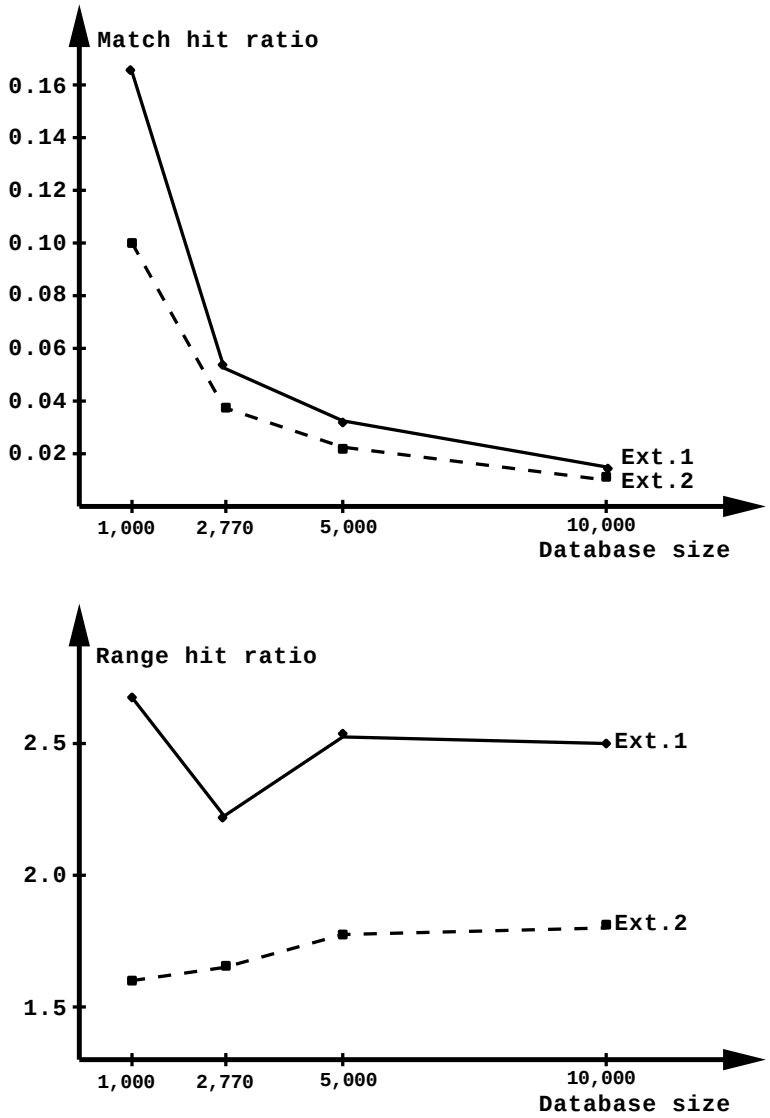



Figure 6.12: The hit ratio vs. data set size for extensions 1 and 2. (File: s2D.mbr, blob size=2KB)

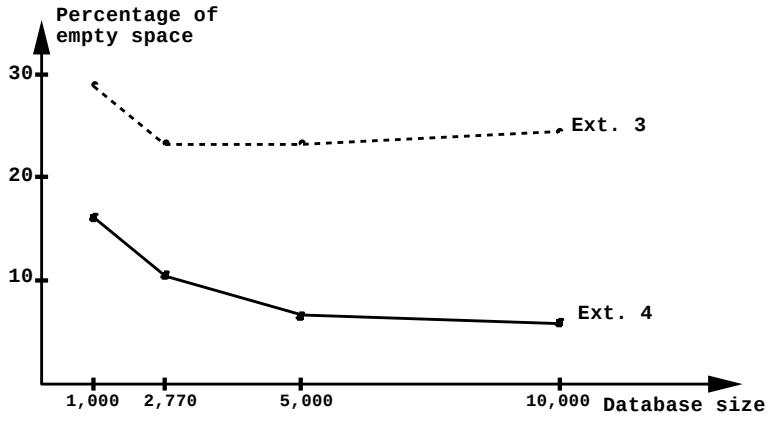



Figure 6.13: Percentage of empty space vs. data set size for extensions 3 and 4. (File: s2D.mbr, blob size=2KB)

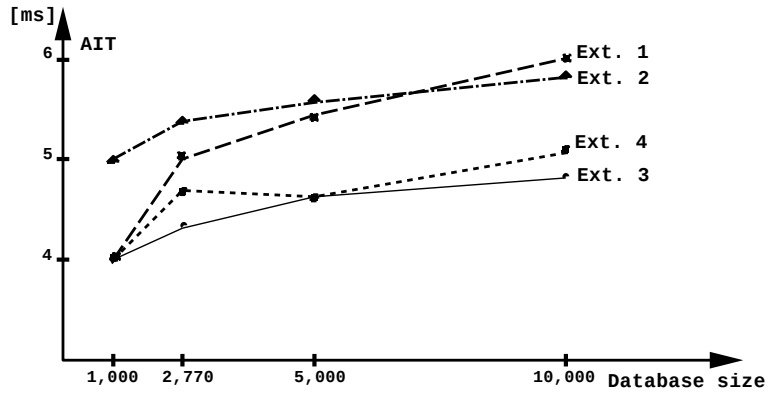



Figure 6.14: AIT vs. data set size for extensions 1, 2, 3, and 4. (File: s2D.mbr, blob size=2KB)

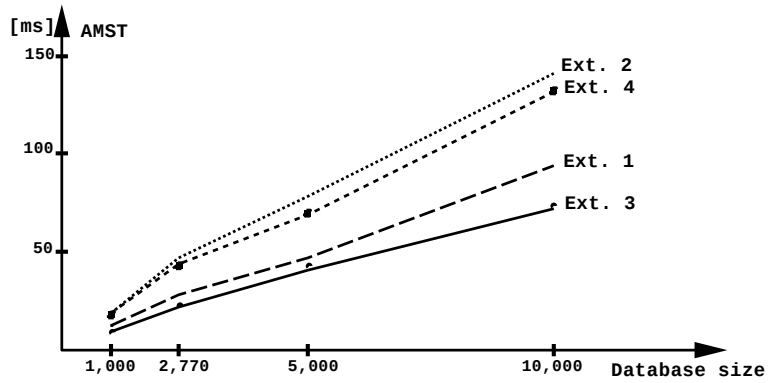



Figure 6.15: AMST vs. data set size for extensions 1, 2, 3, and 4. (File: s2D.mbr, blob size=2KB)

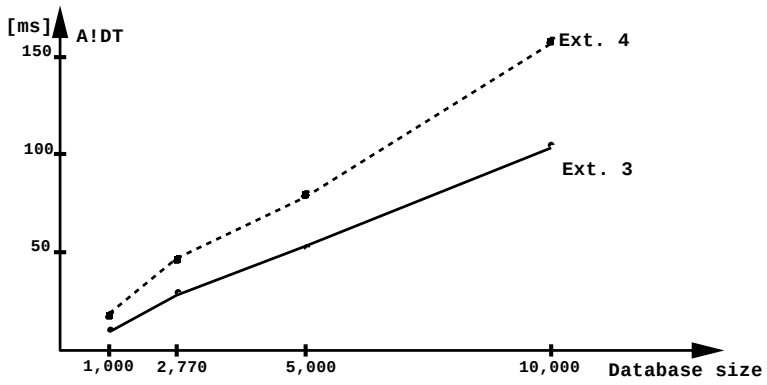



Figure 6.16: A!DT vs. data set size for extensions 3 and 4. (File: s2D.mbr, blob size=2KB)

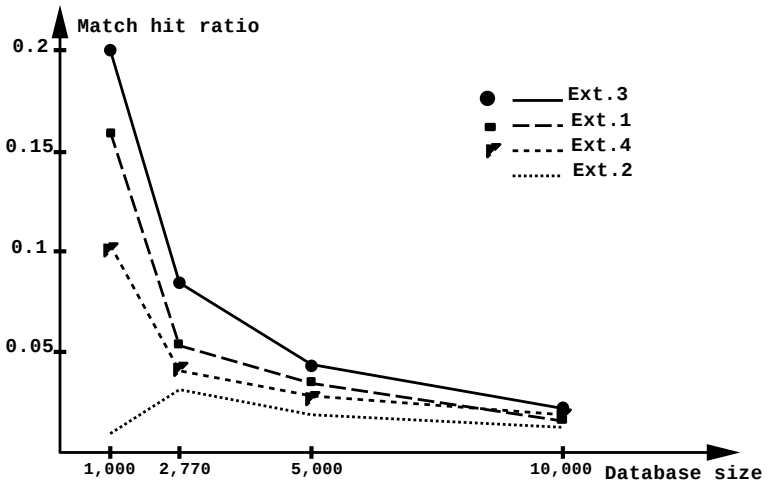



Figure 6.17: The exact match hit ratio vs. data set size for extensions 1, 2, 3, and 4. (File: s2D.mbr, blob size=2KB)

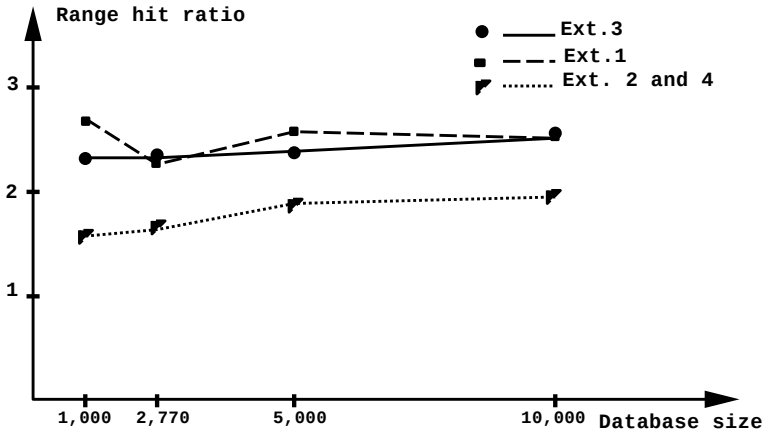



Figure 6.18: The range match hit ratio vs. data set size for extensions 1, 2, 3, and 4. (File: s2D.mbr, blob size=2KB)

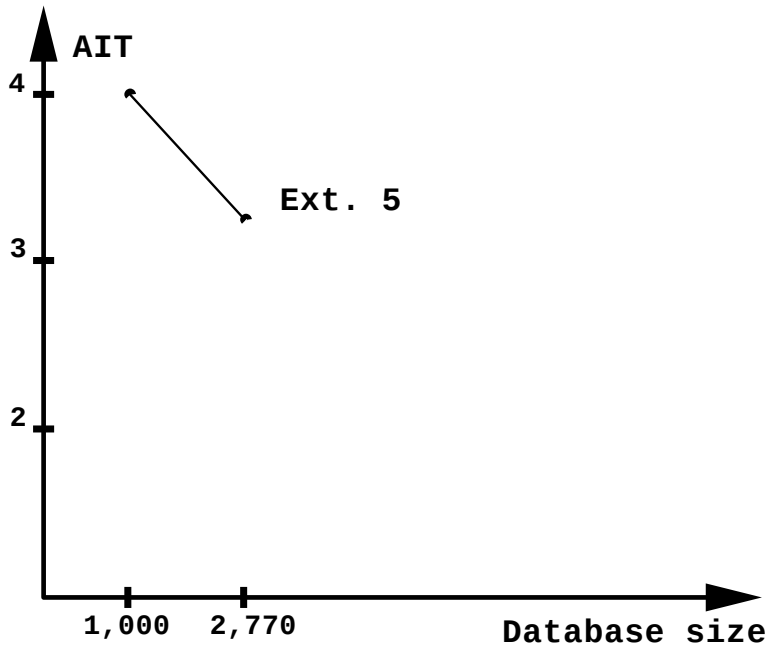



Figure 6.19: AIT vs. data set size for extension 5. (File: rem95.dat, blob size=2KB)

# Chapter 7

# Conclusions and future work

Indexing spatial objects by object position in space is well achieved by SAMs, but spatial objects have also a shape which combined with spatial location define spatial objects more accurately, chapter 2. It is hard to think about a particular object in 2D or 3D geographical space that has not got any area or volume, respectively. This space occupancy of an object defines its shape that has to be included into the key of an index structure. Spatial objects whose position and shape change over time are classed on spatio-temporal objects. Spatio-temporal access methods are actually based on spatial ones. The majority of the spatio-temporal indexing methods take the R_tree structure because it is the simplest access method which can index as many dimensions as needed. If we take three dimensions for the spatial location and shape, and two dimensions for the temporal data, then we can use the same structure, the R_tree, as we have two spatial dimensions and one time dimension. We use a multi-dimensional index structure to support whatever number of dimensions the application needs, and we add new dimensions to the index key to include temporal data of a spatial object.

In this work, we propose a novel approach to building multi-dimensional indexes which attempts to improve the hit ratio by accommodating spatio-temporal search operators within the indexing mechanism. Therefore, we have extended a well known SAM, the R_tree, to support more specific range search operators which are specialized on doing a more restricted task of search. This new approach avoids partitioning of spatial objects.

A second outcome of this work is the developing of a family of methods for indexing into multi-dimensional spatio-temporal data and performing some preliminary testing of these techniques. This result follows directly from our approach mentioned before, that had been refined until obtaining a minimal set of range search operators (presented in chapter 4) to be used as a base to answer spatial, temporal, and spatio-temporal queries.

We also have come to some initial conclusions in terms of the efficiency of these techniques and the sensitivity to parameters such as blob size and the use of redistribution during insertion. These initial conclusions presented in chapter 6 are the substantial improvement in terms of reduction in wasted space through the redistribution policy, and the achievement of better hit ratios and consequently of fewer disk accesses in some range searches.

In general, we have provided support for the usefulness of the TDSO and object-oriented techniques, which are shown in chapters 5 and 6. This was permitted to lead to a very flexible scheme whereby the indexing structure was easily adapted to our problem.

The first problem we mentioned in chapter 1 focusses on avoiding extra geometric calculus when a system is solving a particular spatial query. The majority of the proposed SAMs index objects by a general container like a mbr, which approximates spatial shape and location of the object. But, a spatial query normally needs to answer which objects obey the particular conditions of the query, and an approximation of the location and shape do not achieve these needs. For such cases, it is mandatory to include into the spatial query processor the mentioned geometric calculus to provide the exact set of "answering" objects. We realize that SAMs do not retrieve this set, because of the approximation used as the index key. Thus, we propose to use the R_tree structure extended with specialized range search operators to reduce the cardinality of the set of "retrieved" objects, where will be extracted the other set. We also propose the O_tree that uses an octagon container which is a better approximation of location and shape of a spatial object. The disadvantage of the O_tree is the use of a larger index entry compared to those of the R_tree, without fulfilling the requirement of retrieving the needed set of "answering" objects.

The problem of indexing spatial and temporal attributes using the same access method is solved by treating both characteristics homogeneously. That means we defined some operators to manage both attributes together. However, a user may query a database by any of them separately, and some operators for each kind of query are needed. Thus, we propose spatial, temporal, and spatio-temporal opera-

tors.

A spatial query only uses spatial operators like topological, directional, or proximity operators. These spatial operators are referred to X-Y coordinates which are used to define the approximation included into the index as its key. In our case, we use the mbr.

A temporal query only expresses the question based on temporal operators, which use one time dimension. Even if the application has several time dimensions, we define temporal operators for only one of them in the query. We see that the use of the rectangular approximation (mbr) can be more appropriately applied to a temporal mbr than to a spatial mbr, if time inside each interval is continous, and valid time and transaction time are considered orthogonal. With these assumptions, a temporal mbr defines completely the content of the temporal space permitting the right application of the topological, directional, and proximity operators, already defined to spatial mbrs.

A spatio-temporal query asks for particular conditions for space and time of the object. In this case, we only propose to use topological operators applied to our multi-dimensional container named hyper-rectangle. Because of the location and shape approximation problem, spatio-temporal operators inherit the disadvantage of using hyper-rectangles to retrieve the set of answered objects.

Taking this large set of operators, we realise that we may reduce its cardinality by studying the relationships among them. We propose in chapter 4, a minimal set of operators that can be used as a base to build other operators. With an object-oriented R_tree implementation, we could include the minimal set of operators into the RTree class. We can also extend the class to support whatever behaviour we need for that class in specific applications. Utilizing this object-oriented property, we decide to make six extensions with the minimal changes into the class verifying the advantages of the object-oriented approach.

Our third problem was to search and to choose adequate data to test R_tree extensions and the O_tree proposition. The RTree class of each extension was tested with a particular and reduced set of data objects. We solve this problem partially, due to the lack of available real data. Some particular data sets were generated by following a uniform distribution of the hyper-rectangles in dD space. We realise the lack of an adequate test bed data set to be applied to the extensions.

The generation of a test bed for all of the extensions with several distribution other than uniform is a subject for short term future work. Many of the SAMs reported in the bibliography suffer from the need for more test data because the

behaviour of SAMs depends on the distribution of the data to be indexed. We mentioned this point in chapter 2, e.g. the grid file structure is highly recommended only for uniform data distribution, otherwise the buddy tree is good.

The formal definition of the extension of the D/K model to support spatio-temporal objects is a mid term future work. We partially achieved this point in [BM98], but it remains to implement and test this extension in the implementation of the D/K model. In this extension, we are planning to include temporal operators for more than one time dimension, conforming temporal hyper-rectangles.

Finally, long term future work will involve the inclusion of the RTree class in a more general index to cover 1D and dD management. That is an index method like the grid index where the first scale will be a prefix B_tree, the second scale an R_tree extended with spatio-temporal facilities, and the other scales defined by the user. This structure will be a super index composed of one 1D index and two dD index, where the grid index (dD) contains the others two. We tested this possibility by implementing a grid index where the first scale is a prefix B_tree, and it is presented by E. Paredes in [Par97]. We foresee that this kind of big index will reliably retrieve whatever object is present in the system.

# Bibliography


[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. 'The Object-Oriented Database System Manifesto'. In *Proceedings of the First International Confference on Deductive and Object-Oriented Databases*, pages 25–40. Elsiever, 1989.

[ADF+94] T. Atwood, J. Dubl, G. Ferran, M. Loomis, and L. Wade. *The object database standard: ODMG-93. Release 1.1.* Morgan Kaufmann, first. edition, 1994.

[All83] J. F. Allen. 'Maintaining knowledge about temporal intervals'. *Communications of the A.C.M.*, 26(11):832–843, Nov. 1983.

[Ari86] G. Ariav. 'A temporally oriented data model'. *ACM TODBS*, 11(4):499–527, Dec. 1986.

[Ari87] G. Ariav. 'Design requirements for temporally oriented information systems'. In F. Bodart C. Rolland and M. Leonard, editors, *Proceedings of the IFIP TC 8/WG 8.1. Working Conference on Temporal Aspects in Information Systems*, pages 3–16. North-Holland, 1987.

[BAT90] R. Barrera and K. Al-Taha. 'Models in temporal knowledge representation and temporal DBMS'. Technical paper 90-8, National Center for Geographic Information and Analysis, University of Maine. Orono, ME 04469. USA, October 1990.

[Ben75] J. L. Bentley. 'Multidimensional binary search trees used for associative searching'. *Communications of the ACM.*, 18(9):509–517, 1975.

[Bes84] I. Besembel. 'Estructuras de árbol para la organización de índices'. *SISTEMAS*, 2:1–7, 1984.

[Bes93] I. M. Besembel. 'MIDBS architecture based on a characterization of object-oriented databases, geographic information systems, and multimedia systems architectures'. Draft report. University of Leeds, School of Computer Studies, Leeds,UK, Jan. 1993.

[Bes94] I. M. Besembel. 'El método deductivo MEDEE para analizar, especificar, documentar, diseñar y probar sistemas programados'. *SISTEMAS*, 1994.

[Bes95] I. Besembel. 'Técnica de desarrollo de sistemas de objetos (TDSO)'. Technical report, Escuela de Ingeniería de Sistemas, Universidad de Los Andes, Mérida-Venezuela, 1995.

[BK89] E. Bertino and W. Kim. 'Indexing techniques for queries on nested objects'. *IEEE transactions on knowledge and data engineering*, 1(2):196–214, June 1989.

[BM72] R. Bayer and E. McCreight. 'Organization and maintenance of large ordered indexes'. *Acta informatica*, 1:173–189, 1972.

[BM98] I. Besembel and J. Montilva. 'Modelling spatio-temporal relationships in object-oriented applications'. In M. H. Hanza, editor, *Proceedings of the IASTED International Conference on Artificial Intelligence and Soft Computing. ASC'98.*, pages 242–245. IASTED Acta Press, 1998.

[BMR96] I. Besembel, J. Montilva, and S. Roberts. 'A comparison between R_tree extensions in 2D'. In M. H. Hanza, editor, *Proceedings of the 14th. IASTED International Conference on Applied Informatics*, pages 286–270. IASTED Acta Press, 1996.

[BR97a] I. Besembel and S. Roberts. 'Comparison between multidimensional point and spatial access methods'. In N. Callaos, M. Bennamoun, and J. Aguilar, editors, *Proceedings of the World Multiconference on Systemics, Cybernetics, and Informatics. SCI'97/ISAS'97.*, pages 113–119, 1997.

[BR97b] I. Besembel and S. Roberts. 'Indexing spatio-temporal objects'. In N. Callaos, M. Bennamoun, and J. Aguilar, editors, *Proceedings of the World Multiconference on Systemics, Cybernetics, and Informatics. SCI'97/ISAS'97.*, pages 120–127, 1997.

[BU77] R. Bayer and K. Unterauer. 'Prefix B_trees'. *ACM TODBS*, 2(1):11–26, 1977.

[Bun77] M. Bunge. *Treatise on basic phylosophy. Ontology I: The fourniture of the world.*, volume 3rd. Boston, Reidel, 1977.

[Com79] D. Comer. 'The ubquitous B_tree'. *Computing Surveys*, 11(2):121–137, 1979.

[CR87] J. Clifford and A. Rao. 'A simple general structure for temporal domains'. In F. Bodart C. Rolland and M. Leonard, editors, *Proceedings of the IFIP TC 8/WG 8.1. Working Conference on Temporal Aspects in Information Systems*. North-Holland, 1987.

[Deu90] O. Deux. 'The story of $O_2$'. *IEEE transactions on knowledge and data engineering*, 2(1):91–108, March 1990.

[Duf88] J. F. Dufourd. 'Vers un cadre unique pour spécifier et construire des programmes. Une expérience avec la méthode déductive et les types abstraits algébriques'. *TSI-AFCET*, 7(3), 1988.

[EF88] M. J. Egenhofer and A. U. Frank. 'Designing object-oriented query languages for geographic information systems: human interface aspects'. In *Proc. 3rd. Int. Sympos. on Spatial Data Handling*, 1988.

[EF91] M. Egenhofer and R. Franzosa. 'Point-set topological spatial relations'. *International Journal on Geographical Information Systems*, 5(2):161–174, 1991.

[Ege89] M. Egenhofer. 'A formal definition of binary topological relationships'. In W. Litwin and H. J. Schek, editors, *Proc. of data organization and algorithms*, pages 457–472. Springer Verlag, 1989.

[EH90] M. Egenhofer and J. Herring. ' A mathematical framework for the definition of topological relationships'. In R. Brassel and H. Kishmoto, editors, *Proceedings of the Fourth International Symposium on Spatial Data Handling*, pages 803–813, 1990.

[FB74] R. A. Finkel and J. L. Bentley. 'QuadTrees: A Data Structure for Retrieval on Composite Keys'. *Acta Informatica*, 4:1–9, 1974.

[FB89] A. U. Frank and R. Barrera. 'The Fieldtree: A Data Structure for Geographic Information Systems'. In Buchmann et al., editor, *Design and Implementation of Large Spatial Databases*, pages 29–44. 1st. Symposium, SDD'89, Springer-Verlag, 1989.

[Fra91] A. U. Frank. 'Properties of Geographic Data: Requirements for Spatial Access Methods'. In O. Gunther and H. J. Schek, editors, *Advances in Spatial Databases*, pages 225–233. 2nd. Symposium, SSD'91, Springer-Verlag, 1991.

[Fre89] M. W. Freeston. 'Advances in the design of the BANG file'. In W. Litwin and H. J Schek (Eds.), editors, *Foundations of data organization and algorithms. Proc. 3rd. Int. Conference, FODO 1989.*, pages 322–338. Springer-Verlag, LNCS 367., 1989.

[Fre93] M. Freeston. 'Begriffsverzeichnis: a concept index'. In M. Worboys and A. F. Grundy (Eds.), editors, *Advanced database systems. Proc. 11th. British National Conference on Databases, BNCOD 11.*, pages 1–22. Springer-Verlag. LNCS 696., 1993.

[GB91] O. Gunther and J. Bilmes. 'Tree-Based Access Methods for Spatial Databases: Implementation and Performance Evaluation'. *IEEE Transaction on Knowledge and Data Engineering*, 3(3):342–356, Sep. 1991.

[GHM78] J. V. Guttag, E. Horowitz, and D. R. Musser. 'Abstract data types and software validation'. *A. C. M. Communications*, 21(12), 1978.

[Gun89] O. Gunther. 'The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases'. In IEEE, editor, *IEEE 5th. International Conference on Data Engineering*, pages 598–605, 1989.

[Gut77] J. V. Guttag. Abstract data types and the development of data structures'. *A. C. M. Communications*, 20(6), 1977.

[Gut84] A. Guttman. 'R_trees: A dynamic index structure for spatial searching'. In ACM, editor, *ACM SIGMOD Conference on Management of Data*, pages 47–57, 1984.

[Her90] D. Hernández. Optimización del archivo maya. Proyecto de grado, Escuela de Ingeniería de Sistemas, Universidad de Los Andes. Mérida-5101. Venezuela, April 1990.

[Hug91] J. G. Hughes. *'Object-Oriented databases'*. C.A.R. Hoare series editor. Prentice-Hall, 1991.

[Jor94] M. Jorge. Desarrollo de la librería de classes persistentes en C++. Proyecto de grado, Escuela de Ingeniería de Sistemas, Universidad de Los Andes. Mérida-5101. Venezuela, 1994.

[KGK93] W. Kim, J. Garza, and A. Keskin. 'Spatial data management in database systems: research directions'. In D. Abel and B. Ooi, editors, *Advances in spatial databases*, pages 1–12. Third International Symposium, SSD'93, Springer-Verlag. LNCS 692, 1993.

[Kim90] W. Kim. *'Introduction to Object-Oriented Databases'*. The MIT Press, 1990.

[Kim95] W. Kim. *Modern database systems. The object model, interoperability, and beyond.* ACM Press and Addison-Wesley, first. edition, 1995.

[KL92] W. Kim and F. H. Lochovsky. *'Object-oriented concepts, databases ans applications'*. Academic Press, first. edition, 1992.

[KSSS89] H.-P. Kriegel, M. Schiwietz, R. Schneider, and B. Seeger. 'Performance Comparison of Point and Spatial Access Methods'. In Buchmann et al., editor, *Design and Implementation of Large Spatial Databases*, pages 89–114. 1st. Symposium on Spatial Data Handling, SDD'89, Springer-Verlag, 1989.

[Lan89] G. Langran. 'A review of temporal database research and its use in GIS applications'. *International Journal of Geographical Information Systems*, 3(3):215–232, 1989.

[LT92] R. Laurini and D. Thompson. *'Fundamentals of spatial information systems'*, volume The Apic series. Academic Press, first. edition, 1992.

[MB96] J. Montilva and I. Besembel. 'An object oriented model for representing and manipulating networks in space'. In M. H. Hanza, editor, *Proceedings of the 14th. IASTED International Conference on Applied Informatics*, pages 300–302. IASTED Acta Press, 1996.

[Mon93] J. A. Montilva. *'An integration method applied to the design of a data/knowledge model for multimedia and spatial applications'*. PhD

thesis, School of Computer Studies, University of Leeds. Leeds, LS2 9JT,UK, Jan. 1993.

[MS86] D. Maier and J. Stein. 'Indexing in an Object-Oriented DBMS'. In *IEEE workshop in object-oriented databases*, pages 171–182. IEEE, 1986.

[MS90] D. Maier and J. Stein. 'Development and Implementation of an Object-Oriented Database Management System'. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Management Systems*, pages 167–185. Morgan Kaufman, 1990.

[NA88] S. Navathe and R. Ahmed. 'TSQL: A language interface for history databases'. In C. Rolland et al, editor, *Temporal aspects in information systems*, pages 109–122. IFIP88, North-holland, 1988.

[NHM92] L. Bennett N. Hazelton and J. Masel. 'Topological structures for 4-dimensional geographic information systems'. *Computer Environment and Urban Systems*, 16:227–237, 1992.

[NHS84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. 'The Grid File: An Adaptable, Symmetric Multikey File Structure'. *ACM Transactions on Database Systems*, 9(1):38–71, Mar. 1984.

[OSDM91] B. C. Ooi, R. Sacks-Davis, and K. J. McDonell. 'Spatial indexing in binary decomposition and spatial bounding'. *Information Systems*, 16(2):211–237, 1991.

[Par97] E. Paredes. Desarrollo de la clase índice malla en C++. Proyecto de grado, Escuela de Ingeniería de Sistemas, Universidad de Los Andes. Mérida, 5101, Venezuela, 1997.

[Peu85] D. J. Peuquet. 'A conceptual framework and comparison of spatial data models'. pages 66–113, 1985.

[Ram94] C. Ramirez. Desarrollo de la clase árbol_B$^+$ con prefijos en C++. Proyecto de grado, Escuela de Ingeniería de Sistemas, Universidad de Los Andes. Mérida-5101. Venezuela, 1994.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *'Object-oriented Modeling and Design'*. Prentice Hall, first edition, 1991.

[RGHH91] S. A. Roberts, M. N. Gahegan, J. Hogg, and B. Hoyle. 'Application of object-oriented databases to geographic information systems'. *Information and software technology*, 33(1):38–45, Jan/Feb. 1991.

[Rob81] J. T. Robinson. 'The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes'. In *ACM SIGMOD*, pages 10–18. ACM, 1981.

[Rob95] S. A. Roberts. 'Schema for utilising environmental monitoring data'. Work paper, 1995.

[Sam89] H. Samet. 'Hierarchical Spatial Data Structures'. In Buchmann et al., editor, *Design and Implementation of Large Spatial Databases*, pages 193–212. 1st. Symposium, SDD'89, Springer-Verlag, 1989.

[SK90] B. Seeger and H. P. Kriegel. 'The buddy-tree: An efficient and robust access method for spatial data base systems'. In *Proc. 16th. Conference on Very Large Databases, VLDB.*, pages 590–601, 1990.

[SK91] R. Schneider and H. P. Kriegel. 'The TR*_tree: A new representation of polygonal objects supporting spatial queries and operations'. In H. Bieri and H. Noltemeier, editors, *Proceedings of the International Workshop on Computational Geometry, CG'91*, pages 249–263. Springer-Verlag, 1991.

[SK92] R. Schneider and H. P. Kriegel. 'Indexing the spatiotemporal monitoring of a polygonal object'. In P. Bresnahan et al, editor, *Proceedings of the Fifth International Symposium on Spatial Data Handling*, pages 200–209. Springer-Verlag, 1992.

[SK94] S. Soukeras and P. King. 'Temporal databases: an event oriented approach'. In D. S. Bowers, editor, *Directions in Databases*, pages 38–54. Proceedings of the 12th. British National Conference on Databases, BNCOD12., North-Holland, July 1994.

[Sno92] R. Snodgrass. 'Temporal databases'. In A. U. Frank, I. Campari, and U. Formentini, editors, *'Theories and Methods of Spatio-Temporal Reasoning in Geographic Space'*, pages 22–64. International Conference GIS, Springer-Verlag. LNCS 639, Sep. 1992.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. 'The R$^+$_Tree: A Dynamic Index for Multi-Dimensional Objects'. In *13th. International Conference in Very large Databases*, pages 507–518, 1987.

[SS88] A. Segev and A. Shoshani. 'Modeling temporal semantics'. In C. Rolland et al, editor, *Temporal aspects in information systems*, pages 47–57. IFIP88, North-holland, 1988.

[ST90] D. Straube and M. Tamer. 'Queries and query processing in object-oriented database systems'. *A.C.M. Transactions on Information Systems*, 8(4):387–430, Oct. 1990.

[SW88] H. Samet and R. E. Webber. 'Hierarchical Data Structures and Algorithms for Computer Graphics'. *IEEE Computer Graphics & Applications*, pages 48–68, May. & Jul. 1988.

[SW93] H. W. Six and P. Widmayer. 'Spatial access structures for geometric databases'. In B. Monien and Th. Ottmann, editors, *Data structures and efficient algorithms. Final report on the DFG special joint initiative.*, pages 214–232. Springer-Verlag. LNCS 594., 1993.

[TSP93] M. Tamer, D. Straube, and R. Peters. 'Query processing issues in object-oriented knowledge base systems'. JAI Press, 1993.

[Val96] L. Valet. *'An object-orient approach to the modelling of time-varying spatial data'.* PhD thesis, School of Computer Studies, University of Leeds. Leeds, LS2 9JT, UK, May 1996.

[WK90] M. J. Willshire and H. J. Kim. 'Properties of Physical Storage Models for Object-Oriented Databases'. In *IEEE PARBASE-90*, pages 94–99, 1990.

[XHL90] X. Xu, J. Han, and W. Lu. 'RT_Tree: An improved R_tree index structure for spatiotemporal databases'. In *Proceedings of the Fourth International Symposium on Spatial Data Handling*, pages 1040–1049. Springer-Verlag, 1990.

# Appendix A

# A comparative table of OODBMS

This appendix shows a comparative table between six database management systems (DBMS) which refer to themselves as object-oriented database management systems (OODBMS). This table was built based on the characteristics of OODBMS presented by M. Atkinson et al. in [ABD+89] as the main features that a system must have to qualify as an OODBMS.

These characteristics can be categorized into three groups:

**Mandatory:** the OODBMS must satisfy two types of facilities, those concerned with DBMS and those due to object-oriented systems (OOS). These facilities are persistence, secondary storage management, concurrency, recovery, query, complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility, and computational completeness.

**Optional:** features to be added to the system for improving it. Multiple inheritance, type checking and type inferencing, distribution, design transactions, and versions.

**Open:** features which may be included at the designers dicretion. These include: programming paradigm, representation system, type system, and uniformity.

All of these features are well explained in the aforementioned manifesto and we take it for building our comparative table in a more standard form. More clearly, the manifesto considers complex objects as objects which are built from simpler ones by

| *OODBMS* | *Complex objects* | *Object identity* | *Encapsulation* | *Types and classes* |
|---|---|---|---|---|
| *ORION* | Composite objects. Include multimedia objects. | OID consists of a class id an instance id and eventually a site id. | Classes whose instances are objects, encapsulate data and behavior. | Classes with its extensions can be manipulated at run-time. |

Table A.1: Object and classes.

applying constructors to them. Simple objects are integers, characters, bytes strings of any length, booleans, and floats. Complex object constructors are tuples, sets, bags, lists, arrays, etc. Any constructor should apply to any object. The object identity must support object's existence independent of its value.

Encapsulation provides a form of logical data independence, but in some cases it is not needed. In databases, an object encapsulates both data and program, in other words, structure and behavior. Types and classes are useful for representing real objects and they help code sharing (reusability). Tables A.1 and A.2 illustrate these concepts for the chosen OODBMS.

Overriding permits the redefinition of operations resulting in a single name denoting a set of different programs, this is called overloading. The operation name problem is resolved at run-time, called late binding. The computational completeness express the requirements of the DML of the database system, which has to offer any computable function. Table A.3 shows the particularities of the chosen OODBMS.

Extensibility means the posibility to define new type or classes without any distinction between system defined and user defined. Persistence of database objects should be orthogonal, each object is independent of its type. All of the secondary storage management services are invisible to the user. These services are indexing, clustering, buffering, access path selection, and query optimization. Finally, the three features as concurrency, recovery, and ad hoc query facilities are considered indispensable, and in any case, they have to be supported at least as in the DBMS. Tables A.4 and A.5 show these properties of the chosen OODBMS. In database systems, persistence is treated as a property of the elements in the database. When a program opens a database, it supplies all the variables needed for containing the values obtained from the database. Thus, this program is normally written in some programming language (PL) which has embedded the sentences of the database

| *OODBMS* | *Complex objects* | *Object identity* | *Encapsulation* | *Type and classes* |
|---|---|---|---|---|
| $O_2$ | Yes, regardless of whether they are persistent or not. Complex objects manager support tuples, lists, sets, and multimedia objects. | Yes, but retrieval of objects is made by its name. The OID is the record identifier RID in WiSS. Temporary OIDs are generated for new jects and they are changed before commit. | Classes whose instances are objects and which encapsulate data and behavior and types whose instances are values. Methods can be private or public. | Types (ADT) cannot be modified at run-time. Concepts of 'shadow class' and class are considered separately. |
| *GemStone* | Complex objects like composite objects in ORION. Supported by Smalltalk-80. | Supported by Smalltalk-80. | Classes encapsulate structure and behavior. | Classes with its extensions can be manipulated at run-time. |
| *ONTOS* | Objects have a set of rutines and a set of properties. Unstructured text and graphics. | Different identification between a C++ object and a DB object. Name identification is possible. | Objects encapsulate properties and behavior. | Types (ADT) cannot be modified at run-time. It is strongly typed. |
| *IRIS* | Primitive and composite objects are classified as literal and non-literal. | Not specified. | Types are objects and they encapsulate data and behavior. | Types cannot be modified at run-time. |
| *CACTIS* | Objects and relationships called milestone. | Not specified. | Objects re-encapsulate data and behavior. | It does not support classes |

Table A.2: Object and classes. (Continuation)

| *OODBMS* | *Inheritance. Class or type hierarchies.* | *Overriding, overloading, and late binding.* | *Computational completeness* | *Extensibility* |
|---|---|---|---|---|
| *ORION* | Single and multiple. A class is the root class. | All of them are present. | Supported by C. | Supported by schema evolution. |
| *$O_2$* | Single and multiple. A class called Object is the root class. | Late binding is done in constant time. Information about the inherited method is duplicated down the inheritance hierarchy. Support method overriding. | Supported by $CO_2$ (C) or $BasicO_2$ (Basic). | Extensibility of the system are preserved by the possible schema changes. |
| *GemStone* | Single. A class is the database root. | All of them supported by Smalltalk-80. | Supported by Smalltalk-80, C, and C++. | Extensibility are preserved by dynamic schema modification. |
| *ONTOS* | Single. A class called Entity is the root class. | Supported by run-time invocable functions. | Supported by C and C++. | Not specified. |
| *IRIS* | Single and multiple. | Overloading. | Not specified, but OSQL is embedded in Common LISP. | Supported by extensible types. |
| *CACTIS* | Single and multiple. | These functions are supported via type redefinition at run-time. | Supported by C, Pascal, or Fortran. | New objects can be created and new attribute types too. |

Table A.3: Inheritance and extensibility.

management language (DML) of the database system. Sometimes, the program is written in an integrated database programming language (DBPL), but it also supplies the containers for values obtained from the database. These containers are in main memory in a special space called workspace, where is written all the changes needed to the persistent values which will be actually written to disk when the transaction commits. Here, a transaction (T) is a set of actions -probably only one action- upon the values stored in the database, and it has the atomicity property which is that T is completly executed or not executed at all. The relationship between persistence and concurrency/security in the database area is very important, because this last assures the database consistency from a state change to another.

The optional features include multiple inheritance, where a class can have several superclasses. Type checking and type inferencing are beneficial in OODBMS because integrity constraints are very important. Distribution is very useful today and ideally, the system has to be distributed. Following the comparative table, table A.6 presents these characteristics.

About design transaction and versions, we consider them vital in CAD/CAM and CASE applications. Design transactions include long transactions or nested transactions. In open choices, programming paradigm as logic, functional, or imperative can be used. The representation system is defined by the set of types or classes, which can be extended in different ways. Type system is referred to encapsulation or some other system is used. Table A.7 presents these characteristics.

Finally, uniformity is explained at three levels. First, at the implementation level, one must decide which information should be stored as objects or in other ad hoc form. Secondly, at the programming language level, the problem is centered on types and classes. Finally, at the interface level, the key corresponds to a uniform view of types or classes, objects, and methods. We also include in our table features as: object interface type, the concept of attribute, data definition language, query processing and optimization, clustering, buffering, implementation language, attached predicates, work space, objects format, advantages, and disadvantages. Object interface type is referred to the form of communications between objects. In attribute concept, we consider if the OODBMS contains or not this concept. The name and some little things are included in the data definition language. Table A.8 shows the values considered for these parameters.

In query processing and optimization, we consider a little description of the more important characteristics. Tables A.9 and A.10 present the corresponding values to these characteristics.

| *OODBMS* | *Persistence* | *Secondary storage management* | *Concurrency* | *Recovery* |
|---|---|---|---|---|
| *ORION* | Pages are the basic unit of persistence. A garbage collector. By reachability. | A storage manager which manages disk space and it implements indexes for associative accesses to objects. | Lock the descriptor objects. Timestamp in longdata descriptor object and client page buffer pool. | Only soft crashes and user-initiated transaction abort. |
| $O_2$ | Pages are the basic unit of persistence A circular garbage collector of unreferenced objects. Persistent structures are record-structured sequential files, unstructured files, and long data items. By reachability. Classes with extension are persistent. | Task of the disk manager which is WiSS (Wisconsin storage system). | For "flat" records, it is provided by WiSS. Concurrency on the schema are different to concurrency on objects. For objects, a two-phase locking algorithm on pages and files. | Crash recovery available for classes and methods. Recovery may be switched on or off. Use savepoints. |
| *GemStone* | By reachability A database persistent root. A garbage collector. | Stone is built upon the underlying VMS file system. | Optimistic and pessimistic techniques. Pessimistic is used when conflicts are high or transactions are so long. | Soft and hard crashes. |

Table A.4: Persistence, concurrency, and recovery.

| *OODBMS* | *Persistence* | *Secondary storage management* | *Concurrency* | *Recovery* |
|---|---|---|---|---|
| *ONTOS* | By reachability Objects persist until the users explicitly delete them. Object root called Entity. | An area is implemented in UNIX by a file. Areas decompose in segments. Objects(chunks) is the lowest level of granularity. | It is rudimentary and based on locks. A segment is the unit of transfer between disk and the server memory. | Yes, but it is not specified. |
| *IRIS* | It permits explicit deletion of persistent objects, but guarantee referential integrity. | Based on relational model. | Yes. Relational DB. Via HP-SQL. | All. Via HP-SQL. |
| *CACTIS* | Not specified. | Not specified. | Timestamping at attribute level. Partial sorting. | Lack of a rollback and recovery mechanism. |

Table A.5: Persistence, concurrency, and recovery. (Continuation)

| *OODBMS* | *Ad hoc query facility* | *Multiple inheritance* | *Type checking and type inferencing* | *Distribution* |
|---|---|---|---|---|
| *ORION* | Yes. It is similar to the relational databases. | Yes. | Objects are loaded on demand. | In ORION-2. It runs on a local area network. TCP/IPC sockets adopted. |
| *$O_2$* | Yes, it is possible by the database programming language (DBPL). | Yes. | Objects are loaded on demand. | Ethernet and transport protocols are TCP/IP. For each process running on a workstation, there is a mirror process running on the server. The lock table and the buffer managed by WiSS are shared by all process. |
| *GemStone* | Not specified but it can be supported by OPAL. | Yes, in the new version of Smalltalk. | It is present but not supported by Smalltalk. | Not distributed. |
| *ONTOS* | SQL interface. | Not specified. | Type checking at compile time whenever possible. The user may optionally defer it to run time when necessary. | Client server architecture. Local area network of workstation. |
| *IRIS* | OSQL. | Yes. | Yes. | Not specified. |
| *CACTIS* | Not specified. | Yes. | Yes. | Not distributed. |

Table A.6: Query and distribution.

| *OODBMS* | *Design transactions* | *Versions* | *Representation system* | *Type system* |
|---|---|---|---|---|
| *ORION* | Transactions are serializable. The lock manager mantains a lock table and a blocked transaction table. | Transient and working versions. A version is a specific instance of versioned object. | Class hierarchy contains primitive classes. | Encapsulation. |
| $O_2$ | Support transaction modes which adapt system to execution mode or development mode. | Not specified. | Types appear as components of classes. Types are constructed recursively using the atomic types. | Encapsulation. |
| *GemStone* | Transactions are serializable. | Not specified. | All on the system are objects. | Encapsulation. |
| *ONTOS* | Atomics and bounded by starting and either committing or aborting a transaction. Nested and shared transactions. | Versions of objects and general version configurations. | Types are constructed recursively using the atomic types. | Encapsulation. |
| *IRIS* | Not specified. | Not specified. | By types. | Encapsulation. |
| *CACTIS* | Not specified. | Not specified. | By types. | Encapsulation. |

Table A.7: Transactions, versions, and system type.

| *OODBMS* | *Uniformity* | *Object interface type* | *Attribute concept* | *Data definition language* |
|---|---|---|---|---|
| *ORION* | Methods are objects. | By messages. | Defined. | Lisp Statice, extension to LISP. |
| *$O_2$* | Methods are procedures attached to the objects. Classes, types, and values are not objects. | By messages. Message passing manager. | Yes, it is present in the tuple type. | Schema command interpreter, $CO_2$ compiler, Basic$O_2$ compiler, and OOPE. |
| *GemStone* | Not specified. | By messages. | Set-valued attributes. It can be heterogeneous. | OPAL. Alltalk, extension to Smalltalk. |
| *ONTOS* | Differences between objects, types, and functions. | By functions. | Supported by the abstract class called association. | Extension to C called Type Definition Language (TDL) and C Object processor (COP). |
| *IRIS* | Not specified. | By functions. | It is called functions and may be single-valued or multi-valued with heterogeneous objects. | OSQL embedded in C. |
| *CACTIS* | Not uniform. | By values. | Defined and derived by means of an evaluation rule. Classified as important or not. | Not specified, but it is compiled using subqueries. |

Table A.8: Implementation characteristics.

| *OODBMS* | *Query processing and optimization* | *Clustering* | *Buffering* | *Implementation language* |
|---|---|---|---|---|
| *ORION* | Similar to the relational database,but with more complex statistics. Algorithm is similar to nested loop algorithm in relational databases. | Simple. By class. Object directory with extendible hashing. | A page buffer pool and an object buffer pool with its managers. Dual-buffer evaluation schema. | Two systems. ORION1 single user, multitask system. ORION1X, ORION2 network. Common LISP. C under UNIX. |
| *$O_2$* | A query returns an object or a value. The query language is a subset of the programming language, functional, and first order. | A cluster manager based on information given by the DBA. The clustering algorithm operates at commit time on both, new and old objects. Not composite objects like ORION. | A buffer manager takes care of translating OID into memory addresses Dual buffer management schema. A page buffer (disk format), and an object buffer pool (memory format). | C under UNIX. (SunOS4.0) |
| *GemStone* | A single-target query along a class-composition hierarchy. Query optimization is not supported. | Under DBA responsability. | Not specified. | Smalltalk-80 and C. |

Table A.9: Query processing, clustering, and buffering.

| *OODBMS* | *Query processing and optimization* | *Clustering* | *Buffering* | *Implementation language* |
|---|---|---|---|---|
| *ONTOS* | Query mechanism for identifying a group of logically related objects. | Controlled by programmers. | Controlled by programmers. | C++. |
| *IRIS* | Queries expressed in a tree structure called F_tree. | Yes. Relational Databases. | Yes. By record. | C under UNIX. |
| *CACTIS* | High priority queue without disk accesses. Processes are divided in pieces which are treated independently in a priority queue. | Dynamic, based on object statistics about the actual amount of disk I/O. Object past behavior. | Not specified. | C under UNIX. |

Table A.10: Query processing, clustering, and buffering. (Continuation)

| *OODBMS* | *Attach predicates* | *Work space* | *Objects format* | *Space overhead* |
|---|---|---|---|---|
| *ORION* | Not specified. | Not specified. | Two, a disk format and an in-memory | Minimum. No duplication. |
| $O_2$ | Yes, as methods. | A workspace is an object whose type is a set of objects. It can be persistent. | format. Distinction between objects and values. Object OIDs are different to value OIDs. | Not specified. |
| *GemStone* | Possible. | One per user session. | Not specified. | Minimum. |
| *ONTOS* | Supported by means of triggers. | Notion of working directory. | Two, a DB format and a C++ format. | Not specified. |
| *IRIS* | Yes, as functions. Conjunctives, disjunctive, and non-recursive rules. | Not specified. | Two, a disk format and an in-memory format. | Medium respect to ORION. |
| *CACTIS* | Yes, constraints on attributes. | Not specified. | Not specified. | Not specified. |

Table A.11: Predicates, format, and space.

We consider too, if the system contains attach predicates or not, how it manages the work space and how is objects format. Additionally, we include four characteristics presented by Willshire and Kim in [WK90], which are space overhead, instance modification, indexing, and schema changes or evolution. Space overhead means the quantity of memory needed for storing data. Table A.11 illustrates the mentioned parameters.

Instance modification explains how is done instances updating in the system. The main strategy of indexing is shown as indexing, and schema evolution presents how the schema is updated following classes or methods modification. Table A.12 presents these characteristics.

Finally, table A.13 shows our opinion in some advantages and disadvantages.

| *OODBMS* | *Instance modification* | *Indexing* | *Schema changes or evolution* |
|---|---|---|---|
| *ORION* | At most one write to the DB for the three operation (insertion, deletion and modification. | $B^+$_tree larger class-hierarchy. | Use the deferred update approach. Dynamic schema modification. May require moving instance without a database reorganization. |
| *$O_2$* | Possible, but it is not spacified. | Based on complex objects and inheritance. Class hierarchy indexing with a modification which permit retrieve all objects of a subclass in a single block. | Changes dealing with the structure of a class are disallowed. Methods can be added, dropped, redefined, and classes may be refined into subclasses. A class is deleted only if 1. it has no instances, and 2. no other classes depend on it. |
| *GemStone* | Not specified. | $B^+$_tree index on a collection of objects. Equality or or identity indexes. | Use the immediate update approach. Dynamic schema modification. |
| *ONTOS* | Add a new field to a class definition with existing instances is possible, but it is not dynamically supported. | B_trees and linear hashing. | Superclass change or changing the data type of a field incompatible. |
| *IRIS* | Updating require only one write. The other require several writes. | Yes. Relational DBMS. | By tuple. Dynamic schema modification. |
| *CACTIS* | Not specified. | Not specified. | Little flexibility. |

Table A.12: Instances, indexing, and changes.

| *OODBMS* | *Advantages.* | *Disadvantages.* |
|---|---|---|
| *ORION* | Support versions and changes notification. Transient and working versions. Object identifiers generated by the system. | Objects cannot migrate to another class. Transformation of formats. It does not support either recovery from disk crashes, general triggers, and multiple application programming languages. |
| $O_2$ | Two versions. The workstation version is single-user memory-based, while the server version is multiuser, disk based. Architecture in layers. WiSS bypasses the OS file system. Crash recovery are directly available for classes and methods. Lists or insertable arrays are represented by ordered trees which are very efficient. | When it operate under execution mode, the schema is frozen. It is not possible changes on it. |
| *GemStone* | Indexing and concurrency control are unique. Client/server architecture. | It supports only single inheritance in old versions and not allow objects to be explicitly deleted. |
| *ONTOS* | Client/server architecture. It support parameterized types through an aggregate type. Users can customize both types and properties of types. | It does not support class attibutes or metatypes. Objects can be activated or deactivated by users. |
| *IRIS* | Support rules, long fields and versioning. It is built on top of a conventional relational storage manager (DBMS). Data model has a strongly functional style. Objects may have more than one type. | OSQL does not support either GROUP BY and HAVING clauses on SELECT. It does not support recursive function definitions. |
| *CACTIS* | Integrity constraints are very important in the data model. They are represented by attach predicates and maintaned by a trigger mechanism of out-of-date attributes. | It is not really object-oriented. Interrelation between objects through relationship with two types of connectors. It does not support recovery mechanism, dynamic schema changes, authorization, and security facilities. |

Table A.13: Advantages and disadvantages.

# Appendix B

# Class implementations

---

This appendix includes all of the class implementations utilized in the R_tree extensions. These class implementations are specified unsing the TDSO technique that was described in chapter 4. They are described by their order number already shown in the R_tree universe in the same chapter. Classes (7) ListNode and (8) List only have structure and they were already defined in 4.

| Mar.97 | **Implementation {9}Class ListOf[T]**<br>Classes: T, Integer, void | |
|---|---|---|
| 1 | *Superclasses:* | -**descriptor**: List head. |
| 2 | *Structure:* | -**List()**: Constructors. |
| | descriptor: List | -∼**List()**: Destructor. |
| 3 | *Functions:* | -=(): Mutator. Assignment. |
| | List():ListOf[T] | -**object()**: Observer. Return the actual object. |
| | ∼List() | -**insObjList(), ordInsList()**: Mutators. Insert |
| | =(ListOf[T]):ListOf[T] | a new element before the actual and ordered, |
| | object(): T | respectively. |
| | insObjList(T): T | -**delObjList()**: Mutator. Delete the |
| | delObjList(): T | actual element, otherwise empty list. |
| | objList(): T | -**objList()**: Observer. Return the actual object. |
| | headList(): T | -**headList()**: Observer. Return the head |
| | lastObjList(): T | object and move actual to the head node. |
| | headObjList(): T | -**lastObjList()**: Observer. Move actual |
| | nextList(): T | to the last node. |
| | priorList(): T | -**headObjList()**: Observer. |
| | nextObjList(): T | Return the head object. |
| | priorObjList(): T | -**nextList()**: Mutator. Return the object |
| | cleanList(): T | in the next node moving actual to it. |
| | actualPosList(void): T | -**priorList()**: Mutator. Return the object |
| | whichSens(Integer, | in the anterior node moving actual to it. |
| | Integer): Integer | -**nextObjList()**: Observer. |
| | iPosList(Integer): T | Return the next object. |
| | antItemList(void): T | -**priorObjList()**: Observer. |
| | postItemList(void): T | Return the prior object. |
| | searchList(T): T | -**cleanList()**: Mutator. Clean the list. |
| | numList(): Integer | -**actualPosList()**: Mutator. Move actual to |
| | ordInsList(T):ListOf[T] | the node that contains the searched element. |
| | push(T):ListOf[T] | -**whichSens()**: Observer. Sense calculation. |
| | | -**iPosList()**: Mutator. Move actual i times |
| | | beginning in actual and return the actual object. |
| | | -**antItemList()**: Observer. |
| | | Return the prior object. |
| | | -**postItemList()**: Observer. |
| | | Return the next object. |
| | | -**searchList()**: Observer. Search the object in |
| | | the list moving actual to it. |
| | | -**numList()**: Observer. Return the actual |
| | | number of nodes in the list. |
| | | -**ordInsList()**: Mutator. Ordered insertion. |
| | | -**push()**: Mutator. Stack insertion. |

Figure B.1: TDSO implementation of the **ListOf[T]** class.

| Mar.97 **Implementation {10}Class DKOMError** Classes: Cardinal, Char, Float | | |
|---|---|---|
| 1 | *Superclasses:* | **-c**: Error number. |
| 2 | *Structure:* | **-DKOMError()**: Constructors. |
| | c: Cardinal | **-errorMessage()**: Observer. |
| 3 | *Functions:* | Display the error message. |
| | DKOMError():DKOMError | **-warningMessage()**: Observer. |
| | DKOMError(Cardinal):DKOMError | Display the warning message. |
| | errorMessage(Char) | |
| | warningMessage(Char) | |
| | warningMessage(Char, Char) | |
| | warningMessage(Char, Float) | |
| | tail(Char, Cardinal) | |

Figure B.2: TDSO implementation of the **DKOMError** class.

| Mar.97 **Implementation {11}Class EleType** Classes: Cardinal, Integer, ostream | | |
|---|---|---|
| 1 | *Superclasses:* | **-pn, pi**: Element 1 is |
| 2 | *Structure:* | associated to element 2 |
| | pn: Cardinal | in a node list. |
| | pi: Integer | **-noDef()**: Observer. Define |
| 3 | *Functions:* | an undefined element type. |
| | noDef():EleType | -<<(): Observer. Display |
| | <<(ostream, EleType): ostream | on screen an eletype object. |

Figure B.3: TDSO implementation of the **EleType** class.

Mar.97

**Implementation {12}Class Interval**

Classes: Float, Integer, ostream, istream

| | | |
|---|---|---|
| 1 | *Superclasses:* | **-inf, sup**: Inferior and superior |
| 2 | *Structure:* | bounds of an interval. |
| | inf, sup: Float | **-Interval()**: Constructors. |
| 3 | *Functions:* | -=(): Mutator. Assignment. |
| | Interval(): Interval | -==(): Observer. Return one if |
| | Interval(Float, Float): Interval | both Interval are equal in value. |
| | Interval(Float): Interval | -! =(): Observer. Return one if |
| | =(Interval): Interval | both Interval are different in value. |
| | ==(Interval): Integer | -<(): Observer. Return one if one |
| | ! =(Interval): Integer | Interval is less than the second one. |
| | <(Interval): Integer | -<=(): Observer. Return 1 if one Interval |
| | <=(Interval): Integer | is less or equal than the second one. |
| | >(Interval): Integer | ->(): Observer. Return 1 if one |
| | >=(Interval): Integer | Interval is greater than the second one. |
| | disjoint(Interval): Integer | ->=(): Observer. Return 1 if one Interval |
| | meet(Interval): Integer | is greater or equal than the second one. |
| | inside(Interval): Integer | **-disjoint()**: Observer. Disjointness. |
| | coveredBy(Interval): Integer | **-meet()**: Observer. Meeting. |
| | contain(Interval): Integer | **-inside()**: Observer. Interval 1 is |
| | coverg(Interval): Integer | inside of Interval 2. |
| | overlap(Interval): Integer | **-coveredBy()**: Observer. Coverage. |
| | left(Interval): Integer | **-contain()**: Observer. Containment. |
| | right(Interval): Integer | **-coverg()**: Observer. General coverage. |
| | start(Interval): Integer | **-overlap()**: Observer. Overlapping. |
| | finish(Interval): Integer | **-left(),right()**: Observer. Interval 1 is |
| | adjacent(Interval): Integer | on the left or on the right of Interval 2. |
| | follow(Interval): Integer | **-start(),finish()**: Observer. Interval 1 |
| | precede(Interval): Integer | starts or finishes on the |
| | intersection(Interval): Interval | same value of Interval 2. |
| | display() | **-adjacent(),follow(),precede()**: |
| | displayAsInt() | Observers. Interval 1 is adjacent, follows, |
| | <<(ostream, Interval): ostream | or precedes Interval 2. |
| | >>(istream, Interval): istream | **-intersection()**: Observer. Calculate the |
| | | Interval that is the intersection. |
| | | **-display(),displayAsInt(),<<()**: |
| | | Observers. Display a Interval. |
| | | ->>(): Mutator. Interval input. |

Figure B.4: TDSO implementation of the **Interval** class.

| Mar.97 | **Implementation {13}Class HR**<br>Classes: Interval, Integer, Double, ostream | |
|---|---|---|
| 1 | *Superclasses:* | **-hreg**: Variable array of |
| 2 | *Structure:* | intervals. One per dimension. |
| | hreg: Array[DIM]Of | **-putInt()**: Mutator. Put the |
| | Interval | interval in the specified dimension. |
| 3 | *Functions:* | **-=()**: Mutator. Assignment. |
| | putInt(Integer, | both Interval are equal in value. |
| | Interval): Integer | **-==()**: Observer. Return one if |
| | =(HR): Integer | both hyper-rectangles are equal in value. |
| | ==(HR): Integer | **-!=()**: Observer. Return one if |
| | !=(HR): Integer | one hyper-rectangles are different in value. |
| | <(HR): Integer | **-<()**: Observer. Return 1 if one |
| | <=(HR): Integer | hyper-rectangle is less than the second one. |
| | disjoint(HR): Integer | **-<=()**: Observer. Return 1 if one hyper- |
| | inside(HR): Integer | rectangle is less or equal than the second one. |
| | coveredBy(HR): Integer | **-disjoint()**: Observer. Disjointness. |
| | coverg(HR): Integer | **-inside()**: Observer. Hyper-rectangle 1 |
| | overlap(HR): Integer | is inside of hyper-rectangle 2. |
| | slDisjoint(HR): Integer | **-coveredBy()**: Observer. Coverage. |
| | slMatch(HR): Integer | **-coverg()**: Observer. General coverage. |
| | slCoverg(HR): Integer | **-overlap()**: Observer. Overlapping. |
| | slInside(HR): Integer | **-slDisjoint()**: Observer. Spatial disjointness. |
| | slCoveredBy(HR): Integer | **-slMatch()**: Observer. Spatial matching. |
| | slOverlap(HR): Integer | **-slCoverg()**: Observer. General spatial coverage. |
| | equal(Interval): Integer | **-slInside()**: Observer. Spatial inside. |
| | adjacent(Interval): Integer | **-slCoveredBy()**: Observer. Spatial coverage. |
| | precede(Interval): Integer | **-slOverlap()**: Observer. Spatail Overlapping. |
| | follow(Interval): Integer | **-equal()**: Observer. Temporal equality. |
| | during(Interval): Integer | **-adjacent()**: Observer. Temporal adjacency. |
| | intersection(HR): HR | **-follow()**: Observer. Follow in time dimension. |
| | hRec(HR): HR | **-precede()**: Observer. Temporal precedence. |
| | isAPoint(): Interval | **-during()**: Observer. Temporal inclusion. |
| | volume(): Double | **-intersection()**: Observer. Calculate the |
| | display() | hyper-rectangle that is the intersection. |
| | <<(ostream, HR): ostream | **-hRec()**: Observer. Calculate the hyper- |
| | | rectangle that cover both hyper-rectangles. |
| | | **-isAPoint()**: Observer. Return one if the |
| | | hyper-rectangle is a point. |
| | | **-volume()**: Observer. Return the hyper-area. |
| | | **-display(),<<()**: Observers. HR display. |

Figure B.5: TDSO implementation of the **HR** class.

| Mar.97 | **Implementation {14}Class Blob**<br>Classes: Char, Cardinal, Integer, ostream | |
|---|---|---|
| 1 | *Superclasses:* | **-status**: One if it is actually |
| 2 | *Structure:* | in use. |
| | status, objNum: Char | **-objNum**: Current number |
| | length: Cardinal | of object in it. |
| | left, right: Cardinal | **-length**: Current length in bytes. |
| | tail: Array[TailSize]Of Char | **-left**: Left brother blob number. |
| 3 | *Functions:* | **-right**: Right brother blob number. |
| | Blob():Blob | **-tail**: Blob content. |
| | Blob(Blob):Blob | **-Blob()**: Constructors. |
| | Status():Char | **-Status(), ObjNum(), Length(),** |
| | ObjNum():Char | **-Left(), Right()**: Observers. |
| | Length():Cardinal | Return the current value |
| | Left():Cardinal | for each blob attribute. |
| | Right():Cardinal | **-Tail()**: Observer. Return the |
| | Tail(Integer):Char | required character of the blob tail. |
| | getNew(Integer):Cardinal | **-getNew()**: Observer. Return |
| | display() | a new blob. |
| | fileName(Cardinal): Char | **-display()**: Observer. Display |
| | Status(Char): Integer | a blob. |
| | ObjNum(Char): Integer | **-fileName()**: Observer. Return |
| | Length(Cardinal): Integer | the corresponding file |
| | Left(Cardinal): Integer | name for a given class. |
| | Right(Cardinal): Integer | **-Status(), ObjNum(), Length(),** |
| | Tail(Char, Cardinal): Integer | **-Left(),Right()**: Mutators. |
| | readBlob(Cardinal, Cardinal): Integer | Change the current value in each |
| | writeBlob(Cardinal, Cardinal): Integer | one of the blob attributes. |
| | =(Blob): Blob | **-Tail()**: Mutator. Change the |
| | <<(ostream, Blob): ostream | current value in the required |
| | | character of the tail. |
| | | **-readBlob()**: Observer. Fetch |
| | | the required blob from the |
| | | corresponding file on disk. |
| | | **-writeBlob()**: Mutator. Write |
| | | on disk the specified blob. |
| | | -=(): Mutator. Assignment. |
| | | -<<(): Observer. Display a blob. |

Figure B.6: TDSO implementation of the **Blob** class.

| Mar.97 | **Implementation {15}Class Entry**<br>Classes: HR, Cardinal, Integer, Char, MPoint, Interval, ostream | |
|---|---|---|
| 1 | *Superclasses:* | **-hr**: Entry's hyper-rectangle. |
| 2 | *Structure:* | **-bn**: Blob number associated. |
| | hr: HR | **-Entry()**: Constructors. |
| | bn: Cardinal | **-Hr(), Bn()**: Observers. Return the |
| 3 | *Functions:* | current values of the entry attributes. |
| | Entry():Entry | -==(): Observer. Return 1 if both |
| | Entry(HR, Cardinal):Entry | entries are equal in value. |
| | Entry(Entry):Entry | -!=(): Observer. Return 1 if both |
| | Entry(Char):Entry | entries are different in value. |
| | Hr():HR | **-disjoint()**: Observer. Return the |
| | Bn():Cardinal | blob number if the entry's |
| | ==(Entry):Integer | hyper-rectangle is disjoint. |
| | !=(Entry):Integer | **-match()**: Observer. Return the blob |
| | disjoint(HR):Cardinal | number if entry's hyper-rectangle is equal. |
| | match(HR):Cardinal | **-inside()**: Observer. Return the |
| | inside(HR): Cardinal | blob number if the entry's |
| | cover(HR): Cardinal | hyper-rectangle is inside. |
| | coverg(HR): Cardinal | **-cover()**: Observer. Return the blob |
| | covers(MPoint): Cardinal | number if the entry's hyper-rectangle |
| | overlap(HR): Cardinal | is covered, but it is not inside. |
| | overlap(MPoint): Cardinal | **-coverg()**: Observer. Return the |
| | slDisjoint(HR): Cardinal | blob number if the entry's |
| | slMatch(HR): Cardinal | hyper-rectangle is covered. |
| | slCoverg(HR): Cardinal | **-covers()**: Observer. Return the |
| | slInside(HR): Cardinal | blob number if the entry's |
| | slOverlap(HR):Cardinal | hyper-rectangle covers the point. |
| | slCover(HR):Cardinal | **-overlap()**: Observer. Return the |
| | overlapD(HR, Cardinal):Cardinal | blob number if the entry's |
| | leftD(HR, Cardinal):Cardinal | hyper-rectangle overlaps. |
| | rightD(HR, Cardinal):Cardinal | **-overlap()**: Observer. Return the |
| | leftOverlap(HR, Cardinal):Cardinal | blob number if the entry's |
| | rightOverlap(HR, Cardinal):Cardinal | hyper-rectangle overlaps the point. |
| | tMeet(HR, Cardinal):Cardinal | **-slDisjoint()**: Observer. Spatial disjoint- |
| | equal(HR, Cardinal):Cardinal | ness. Return the blob number if the spatial |
| | start(HR, Cardinal):Cardinal | hyper-rectangle of the entry is disjoint. |
| | finish(HR, Cardinal):Cardinal | **-slMatch()**: Observer. Spatial equality. |
| | adjacent(HR, Cardinal):Cardinal | **-slCoverg()**: Observer. Spatial coverage. |
| | precede(HR, Cardinal):Cardinal | **-slInside()**: Observer. Spatial containment. |
| | follow(HR, Cardinal):Cardinal | **-slOverlap()**: Observer. Spatial overlapping. |
| | during(HR, Cardinal):Cardinal | **-slCover()**: Observer. Spatial coverage only. |

Figure B.7: TDSO implementation of the **Entry** class.

| **Implementation {15}Class Entry** | |
|---|---|
| during(Interval):Cardinal<br>tOverlap(MPoint):Cardinal<br>display()<br>=(Entry): Entry<br>Hr(HR)<br>Bn(Cardinal)<br><<(ostream, Entry): ostream | -**overlapD**(): Observer. Overlapping in the given dimension, returning the blob number.<br>-**leftD**(): Observer. Return the blob number if the hyper-rectangle's dimension is on the left.<br>-**rightD**(): Observer. Return the blob number if the hyper-rectangle's dimension is on the right.<br>-**leftOverlap**(): Observer. Return the blob number if the hyper-rectangle's dimension overlaps on the left.<br>-**rightOverlap**(): Observer. Return the blob number if the hyper-rectangle's dimension overlaps on the right.<br>-**tMeet**(): Observer. Return the blob number if the hyper-rectangle's time dimension meets.<br>-**equal**(): Observer. Return the blob number if the hyper-rectangle's time dimension is equal.<br>-**start**(): Observer. Return the blob number if the hyper-rectangle's time dimension starts at the same point.<br>-**finish**(): Observer. Return the blob number if the hyper-rectangle's time dimension finishes at the same point.<br>-**adjacent**(): Observer. Time adjacency.<br>-**precede**(): Observer. Time precedence.<br>-**follow**(): Observer. Time consequence.<br>-**during**(): Observer. Time containment.<br>-**tOverlap**(): Observer. Time overlapping.<br>-**display**(): Observer. Display the entry.<br>-=(): Mutator. Assignment.<br>-**Hr**(), **Bn**(): Mutators. Change the actual values of the entry's attributes.<br>-<<(): Observer. Display a blob. |

Figure B.8: TDSO implementation of the **Entry** class. (Continuation)

| Mar.97 | **Implementation {16}Class RTreeNode**<br>Classes: Cardinal, Char, Entry, Integer, ostream | |
|---|---|---|
| 1 | *Superclasses:* | **-blobId**: Blob identifier. |
| 2 | *Structure:* | **-level**: Node high in the tree. |
| | blobId: Cardinal | **-entries**: Current number of entries. |
| | level, entries: Char | **-length**: Current length in bytes. |
| | length: Cardinal | **-left**: Left brother node number. |
| | left, right: Cardinal | **-right**: Right brother node number. |
| | rest: Array[MaxEntryMem]Of Entry | **-rest**: Node's entries. |
| 3 | *Functions:* | **-RTreeNode()**: Constructors. |
| | RTreeNode(): RTreeNode | **-maxHb()**: Observer. Return the |
| | RTreeNode(Cardinal, Char, Char, | hyper-rectangle that covers all |
| | (Cardinal, Cardinal, Cardinal): | of the current hyper-rectangles. |
| | RTreeNode | **-=()**: Mutator. Assignment. |
| | RTreeNode(Cardinal):RTreeNode | **-insNewEntry()**: Mutator. Insert |
| | RTreeNode(RTreeNode):RTreeNode | a new entry into the node. |
| | maxHb():HR | **-delExEntry()**: Mutator. Delete |
| | =(RTreeNode):RTreeNode | an existing entry into the node. |
| | insNewEntry(Entry, Cardinal):Integer | **-writeNodeOnDisk()**: Mutator. |
| | delExEntry(Cardinal):Integer | Write on disk the modified node. |
| | writeNodeOnDisk(Cardinal): Integer | **-readNodeFromDisk()**: Mutator. |
| | readNodeFromDisk(Cardinal, Cardinal): | Read from disk the specified node. |
| | Integer | **-cleanR()**: Mutator. Clean all of |
| | cleanR(Cardinal, Cardinal) | the node of the R_tree. |

Figure B.9: TDSO implementation of the **RTreeNode** class.

| Apr.96 | | |
|---|---|---|
| | **Implementation {17}Class RTreeExt1** | |
| | Classes: Char, Cardinal, Float, RTreeNode, Integer, HR, ListOf[T], Entry, EleType | |
| 1 | *Superclasses:* | **-dim,height,entLength,xDim,nCoord**: |
| 2 | *Structure:* | Dimension, current height, length of each |
| | dim,height,entLength, | entry in bytes, position of X coordinate |
| | xDim,nCoord: Char | and number of coordinates, respectively. |
| | entNum,blobRoot,blobNum,bnf, | **-entNum,blobRoot,blobNum,bnf,** |
| | blobSize,insNum,delNum, | **blobSize,insNum,delNum,maQNum,** |
| | maQNum,raQNum: Cardinal | **raQNum**: Actual number of entries, root |
| | emptyPer,tInsTime,tDelTime, | blob number, actual number of blobs in use, |
| | tMaQTime,tRaQTime: Float | actual number of assigned blobs, blob size |
| | root: RTreeNode | in bytes, actual number of insertions, |
| 3 | *Functions:* | actual number of deletions, actual number |
| | RTreeExt1(): RTreeExt1 | of exact match and range queries, resp. |
| | RTreeExt1(Cardinal): RTreeExt1 | **-emptyPer,tInsTime,tDelTime,** |
| | ∼RTreeExt1() | **tMaQTime,tRaQTime**: Current |
| | Dim(),Height(),EntLength(),XDim(), | average of empty space within nodes, |
| | NCoord(): Char | total insertion time, total deletion |
| | EntNum(),BlobNum(),Bnf(), | time, total exact match query time, |
| | BlobSize(),InsNum(),DelNum(), | total range query time, respectively. |
| | MaQNum(),RaQNum(): Cardinal | **-root**: R_tree's root in main memory. |
| | EmptyPer(),avgInsTime(), | **-RTreeExt1()**: Constructors. |
| | avgDelTime(),avgMaQTime(), | **-∼RTreeExt1()**: Destructor. |
| | avgRaQTime(): Float | **-Dim(),Height(),EntLength(),XDim(),** |
| | isMultidim(): Integer | **NCoord(),EntNum(),BlobNum(),Bnf(),** |
| | match(HR,ListOf[Cardinal],Cardinal, | **BlobSize(),InsNum(),DelNum(),** |
| | Cardinal): Float | **MaQNum(),RaQNum(),EmptyPer(),** |
| | range(HR,ListOf[Cardinal],Cardinal, | **avgInsTime(),avgDelTime(),** |
| | Cardinal): Float | **avgMaQTime(),avgRaQTime()**: |
| | insertEnt(Entry,Float,Cardinal): | Observers. Each function permits to see |
| | RTreeExt1 | the value stored in each R_tree attribute. |
| | deletEnt(Entry,Float,Cardinal): | **-isMultidim()**: Observer. Return 1 if |
| | RTreeExt1 | the R_tree has many dimensions. |
| | cleanRTree(Cardinal): RTreeExt1 | **-match()**: Observer. Execute an exact |
| | chooseLeaf(RTreeNode,Entry, | match query returning the number of disk |
| | ListOf[EleType],Cardinal) | accesses, the number of retrieved objects, |
| | linearPickSeeds(RTreeNode,Cardinal) | and the time used to answer the query. |
| | splitNode(RTreeNode,RTreeNode) | **-range()**: Observer. Execute a range |
| | adjustTree(RTreeNode,RTreeNode, | query returning the number of disk |
| | ListOf[EleType],Cardinal) | accesses, retrieved objects, and time. |

Figure B.10: TDSO implementation of the **RTreeExt1** class.

| **Implementation {17}Class RTreeExt1** | |
|---|---|
| findLeaf(RTreeNode,RTreeNode,Entry, Cardinal,ListOf[EleType],Cardinal)<br>condenseTree(RTreeNode, ListOf[EleType],Cardinal)<br>redistribution(RTreeNode,RTreeNode, RTreeNode,RTreeNode,Integer)<br>redistribution2(RTreeNode,RTreeNode, RTreeNode,Integer,Cardinal)<br>deletion(RTreeNode,RTreeNode, RTreeNode,RTreeNode,Integer, Cardinal)<br>deletion2(RTreeNode,RTreeNode, RTreeNode,Integer,Cardinal, Cardinal)<br><<(ostream,RTree)<br>displayNode(Cardinal) | **-insertEnt()**: Mutator. Insert a new entry returning the time used.<br>**-deletEnt()**: Mutator. Delete an existing entry returning the time used.<br>**-cleanRTree()**: Mutator. Delete all of the objects and nodes of the R_tree.<br>**-chooseLeaf()**: Observer. Choose a leaf to store the new entry.<br>**-linearPickSeeds()**: Observer. Pick up the splitting point of a node.<br>**-splitNode()**: Mutator. Split a node following the splitting policy.<br>**-adjustTree()**: Mutator. Change the hyper-rectangle of the needed entries.<br>**-findLeaf()**: Observer. Find the leaf where is the entry that will be deleted.<br>**-condenseTree()**: Mutator. Change hyper-rectangles and delete nodes if needed.<br>**-redistribution()**: Mutator. Redistribute entries among three nodes.<br>**-redistribution2()**: Mutator. Redistribute entries among two nodes.<br>**-deletion()**: Mutator. Delete a node when it has two brother nodes.<br>**-deletion2()**: Mutator. Delete a node when it has only one brother node.<br>**-<<()**: Observer. Display on screen an RTreeExt1 object.<br>**-displayNode()**: Observer. Display on screen the demanded node. |

Figure B.11: TDSO implementation of the **RTreeExt1** class. (Continuation)

Apr.96

**Implementation {17}Class RTreeExt3**

Classes: Character, Float, Integer, Cardinal, EleType, ListOf, Interval, Entry, RTreeNode

| | | |
|---|---|---|
| 1 | *Superclasses:* | **-dim**: Number of dimensions. |
| 2 | *Structure:* | **-height**: Current height. |
| | dim,height,entLength,xDim, | **-entLength**: Length of each entry. |
| | timeDim,nTime,nCoord: | **-xDim**: X coordinate position. |
| | Character | **-timeDim**: Time position. |
| | entNum,blobRoot,blobNum, | **-nTime**: Number of time dimensions. |
| | bnf,blobSize,insNum,delNum, | **-nCoord**: Number of coordinates. |
| | diQNum,nDQNum,maQNum, | **-entNum**: Current entries number. |
| | inQNum,coQNum,slDiQNum, | **-blobRoot**: Blob root number. |
| | slMaQNum,slInQNum, | **-blobNum**: Current number of blobs in use. |
| | slCoQNum,nDDQNum,lDQNum, | **-bnf**: Current number of assigned blobs. |
| | rDQNum,tmQNum,eqQNum, | **-blobSize**: Current blob size. |
| | stQNum,fiQNum,adQNum, | **-insNum**: Actual number of insertions. |
| | preQNum,foQNum,duQNum: | **-delNum**: Actual number of deletions. |
| | Cardinal | **-diQNum,nDQNum,maQNum,inQNum,** |
| | emptyPer,deltaTime,delta, | **coQNum**: Actual number of |
| | tInsTime,tDelTime,tDiQTime, | disjoint, not disjoint, exact |
| | tNDQTime,tMaQTime,tInQTime, | match, inside, and cover dim- |
| | tCoQTime,tSlDiQTime, | dimensional queries. |
| | tSlMaQTime,tSlInQTime, | **-slDiQNum,slMaQNum,slInQNum,** |
| | tSlCoQTime,tNDDQTime, | **slCoQNum**: Current number of |
| | tLDQTime,tRDQTime,tTmQTime, | disjoint, exact match, inside, and |
| | tEqQTime,tStQTime,tFiQTime, | cover queries (Spatial). |
| | tAdQTime,tPreQTime, | **-nDDQNum,lDQNum,rDQNum**: |
| | tFoQTime,tDuQTime: Float | Actual number of not disjoint |
| | root: RTreeNode | in dimension, left in dimension, |
| 3 | *Functions:* | right in dimension queries (1D). |
| | RTreeExt3():RTreeExt3 | **-tmQNum,eqQNum,stQNum,** |
| | RTreeExt3(Cardinal): | **fiQNum,adQNum,preQNum,** |
| | RTreeExt3 | **foQNum,duQNum**: Current number |
| | ~RTreeExt3() | of temporal meeting, equal, start, |
| | Dim(),Height(),EntLength(), | finish, adjacent, precede, follow, |
| | XDim(),TimeDim(),NTime(), | and during temporal queries. |
| | NCoord(): Character | **-emptyPer,deltaTime,delta**: |
| | EntNum(),BlobNum(),Bnf(), | Actual percentage of empty |
| | BlobSize(),InsNum(), | space within nodes, unit of |
| | DelNum(),DiQNum(),NDQNum(), | time to be used in temporal |
| | MaQNum(),InQNum(),CoQNum(), | queries, and unit of space to |
| | SlDiQNum(),SlMaQNum(), | be used in near/far queries. |

Figure B.12: TDSO implementation of the **RTreeExt3** class.

| **Implementation {17}Class RTreeExt3** | |
|---|---|
| SlInQNum(),SlCoQNum(),<br>NDDQNum(),LDQNum(),<br>RDQNum(),TmQNum(),EqQNum(),<br>StQNum(),FiQNum(),AdQNum(),<br>PreQNum(),FoQNum(),<br>DuQNum(): Cardinal<br>isMultidim():Integer<br>EmptyPer(),DeltaTime(),<br>Delta(),avgInsTime(),<br>avgDelTime(),avgDiQtime(),<br>avgNDQTime(),avgMaQTime(),<br>avgInQTime(),avgCoQTime(),<br>avgSlDQTime(),avgSlMaQTime(),<br>avgSlInQTime(),<br>avgSlCoQTime(),<br>avgNDDQTime(),avgLDQTime(),<br>avgRDQTime(),avgTmQTime(),<br>avgEqQTime(),avgStQTime(),<br>avgFiQTime(),avgAdQTime(),<br>avgPreQTime(),avgFoQTime(),<br>avgDuQTime():Float<br>disjoint(HR,ListOf[Cardinal],<br>Cardinal,Cardinal):Float<br>nDisjoint(HR,ListOf[Cardinal],<br>Cardinal,Cardinal):Float<br>match(HR,ListOf[Cardinal],<br>Cardinal,Cardinal):Float<br>inside(HR,ListOf[Cardinal],<br>Cardinal,Cardinal):Float<br>cover(HR,ListOf[Cardinal],<br>Cardinal,Cardinal):Float<br>slDisjoint(HR,ListOf[Cardinal],<br>Cardinal,Cardinal):Float<br>slNDisjoint(HR,ListOf[Cardinal],<br>Cardinal,Cardinal):Float<br>slMatch(HR,ListOf[Cardinal],<br>Cardinal,Cardinal):Float<br>slInside(HR,ListOf[Cardinal],<br>Cardinal,Cardinal):Float<br>slCover(HR,ListOf[Cardinal],<br>Cardinal,Cardinal):Float | **-tInsTime,tDelTime,tDiQTime,**<br>**tNDQTime,tMaQTime,tInQTime,**<br>**tCoQTime**: Current cumulative<br>insertion, deletion, disjoint, not<br>disjoint, exact match, inside, and<br>cover query time.<br>**-tSlDiQTime,tSlMaQTime,**<br>**tSlInQTime,tSlCoQTime**:<br>Current cumulative spatial disjoint,<br>exact match, inside, and cover<br>query time.<br>**-tNDDQTime,tLDQTime,**<br>**tRDQTime**:Current cumulative<br>1D not disjoint in dimension, left<br>in dimension, and right in<br>dimension query time.<br>**-tTmQTime,tEqQTime,tStQTime,**<br>**tFiQTime,tAdQTime,tPreQTime,**<br>**tFoQTime,tDuQTime**: Current<br>cumulative temporal meeting, equal,<br>start, finish, adjacent, precede,<br>follow, and during query time.<br>**-root**: The R_tree root is<br>always in main memory.<br>**-RTreeExt3()**: Constructors.<br>**-∼RTreeExt3()**: Destructor.<br>**-Dim(),Height(),EntLength(),**<br>**XDim(),TimeDim(),NTime(),**<br>**NCoord(),EntNum(),BlobNum(),**<br>**Bnf(),BlobSize(),InsNum(),**<br>**DelNum(),DiQNum(),NDQNum(),**<br>**MaQNum(),InQNum(),CoQNum(),**<br>**SlDiQNum(),SlNDiNum(),**<br>**SlMaQNum(),SlInQNum(),**<br>**SlCoQNum(),NDDQNum(),**<br>**LDQNum(),RDQNum(),TmQNum(),**<br>**EqQNum(),StQNum(),FiQNum(),**<br>**AdQNum(),PreQNum(),FoQNum(),**<br>**DuQNum()**: Observers.<br>Each function returns the current<br>value stored in the attribute. |

Figure B.13: TDSO implementation of the **RTreeExt3** class. (Continuation)

| **Implementation 17 Class RTreeExt3** | |
|---|---|
| nDisjointD(HR,ListOf[Cardinal], Cardinal,Cardinal,Cardinal):Float<br>leftD(HR,ListOf[Cardinal],Cardinal, Cardinal,Cardinal):Float<br>rightD(HR,ListOf[Cardinal], Cardinal,Cardinal,Cardinal):Float<br>tMeet(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>equal(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>start(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>finish(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>adjacent(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>precede(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>follow(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>during(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>insertEnt(Entry,Float,Cardinal): RTreeExt3<br>deletEnt(Entry,Float,Cardinal): RTreeExt3<br>cleanRTree(Cardinal,Cardinal): RTreeExt3<br>chooseLeaf(RTreeNode,Entry, ListOf[EleType],Cardinal)<br>linearPickSeeds(RTreeNode,Cardinal)<br>splitNode(RTreeNode,RTreeNode)<br>adjustTree(RTreeNode,RTreeNode, ListOf[EleType],Cardinal)<br>findLeaf(RTreeNode,RTreeNode,Entry, Cardinal,ListOf[EleType],Cardinal)<br>condenseTree(RTreeNode, ListOf[EleType],Cardinal)<br>redistribution(RTreeNode,RTreeNode, RTreeNode,RTreeNode,Integer) | **-isMultidim()**: Observer. Return 1 if the R_tree has many dimensions.<br>**-EmptyPer(),DeltaTime(),Delta(), avgInsTime(),avgDelTime(),avgDiQtime(), avgNDQTime(),avgMaQTime(), avgInQTime(),avgCoQTime(), avgSlDQTime(),avgSlMaQTime(), avgSlInQTime(),avgSlCoQTime(), avgNDDQTime(),avgLDQTime(), avgRDQTime(),avgTmQTime(), avgEqQTime(),avgStQTime(), avgFiQTime(),avgAdQTime(), avgPreQTime(),avgFoQTime(), avgDuQTime()**: Observers. Each function returns the corresponding attribute value.<br>**-disjoint(),nDisjoint(),match(), inside(),cover(),slDisjoint(), slMatch(),slInside(),slCover(), nDisjointD(),leftD(),rightD(), tMeet(),equal(),start(),finish(), adjacent(),precede(),follow(), during()**: Observers. Each function execute a named query returning the number of disk accesses, number of retrieved objects, and time.<br>**-insertEnt()**: Mutator. Insert a new entry returning the time used.<br>**-deletEnt()**: Mutator. Delete an existing entry returning the time used.<br>**-cleanRTree()**: Mutator. Delete all of the objects and nodes of the R_tree.<br>**-chooseLeaf()**: Observer. Choose a leaf to store the new entry.<br>**-linearPickSeeds()**: Observer. Pick up the splitting point of a node.<br>**-splitNode()**: Mutator. Split a node following the splitting policy.<br>**-adjustTree()**: Mutator. Change the hyper-rectangle of the needed entries.<br>**-findLeaf()**: Observer. Find the leaf |

Figure B.14: TDSO implementation of the **RTreeExt3** class. (Continuation)

| **Implementation {17}Class RTreeExt3** | | |
|---|---|---|
| | redistribution2(RTreeNode,RTreeNode, RTreeNode,Integer,Cardinal) | where is the entry that will be deleted. |
| | deletion(RTreeNode,RTreeNode, RTreeNode,RTreeNode,Integer, Cardinal) | **-condenseTree()**: Mutator. Change hyper-rectangles and delete nodes if needed. |
| | deletion2(RTreeNode,RTreeNode, RTreeNode,Integer,Cardinal, Cardinal) | **-redistribution()**: Mutator. Redistribute entries among three nodes. |
| | <<(ostream,RTree) | **-redistribution2()**: Mutator. Redistribute entries among two nodes. |
| | displayNode(Cardinal) | **-deletion()**: Mutator. Delete a node when it has two brother nodes. |
| | | **-deletion2()**: Mutator. Delete a node when it has only one brother node. |
| | | -<<(): Observer. Display on screen an RTreeExt1 object. |
| | | **-displayNode()**: Observer. Display on screen the demanded node. |

Figure B.15: TDSO implementation of the **RTreeExt3** class. (Continuation)

| Mar.97 **Implementation {18}Class MPoint** Classes: Float, Integer, HR, Interval, ostream | | |
|---|---|---|
| 1 | *Superclasses:* | **-p**: Variable array of |
| 2 | *Structure:* | float. One per dimension. |
| | p: Array[DIM]Of Float | -=(): Mutator. Assignment. |
| 3 | *Functions:* | -==(): Observer. Return one if |
| | =(MPoint): MPoint | both hyper-points are equal in value. |
| | ==(MPoint): Integer | -!=(): Observer. Return one if |
| | !=(MPoint): Integer | one hyper-points are different in value. |
| | <(MPoint): Integer | -<(): Observer. Return 1 if one multi- |
| | <=(MPoint): Integer | dimensional point is less than the second one. |
| | >(MPoint):Integer | -<=(): Observer. Return 1 if one multi- |
| | >=(MPoint):Integer | dimensional point is less or equal than the other. |
| | disjoint(HR): Integer | ->(): Observer. Return 1 if one multi- |
| | meet(HR):Integer | dimensional point is greater than the second one. |
| | inside(HR): Integer | ->=(): Observer. Return 1 if one multi- |
| | coveredBy(HR): Integer | dimensional point is greater or equal than the other. |
| | slDisjoint(HR): Integer | **-disjoint()**: Observer. Disjointness. |
| | slOverlap(HR): Integer | **-meet()**: Observer. Meeting. |
| | slInside(HR): Integer | **-inside()**: Observer. Multi-dimensional point |
| | slMeet(HR): Integer | is inside of hyper-rectangle . |

Figure B.16: TDSO implementation of the **HR** class.

| Mar.97 | **Implementation 18 Class MPoint** | |
|---|---|---|
| | during(Interval): Integer | **-coveredBy()**: Observer. Coverage. |
| | equivalent(Interval): Integer | **-slDisjoint()**: Observer. Spatial disjointness. |
| | adjacent(Interval): Integer | **-slOverlap()**: Observer. Spatial overlapping. |
| | follow(Interval): Integer | **-slInside()**: Observer. Spatial inside. |
| | precede(Interval): Integer | **-slMeet()**: Observer. Spatial meeting. |
| | tOverlap(HR): Integer | **-during()**: Observer. Temporal inclusion. |
| | start(Interval): Integer | **-equivalent()**: Observer. Temporal equality. |
| | finish(Interval): Integer | **-adjacent()**: Observer. Temporal adjacency. |
| | hRec(HR): HR | **-follow()**: Observer. Temporal consequence. |
| | hrfmp(): HR | **-precede()**: Observer. Temporal precedence. |
| | display() | **-tOverlap()**: Observer. Temporal overlapping. |
| | <<(ostream, MPoint): ostream | **-start()**: Observer. Time starts on the same point. |
| | | **-finish()**: Observer. Time finishes on the same point. |
| | | **-hRec()**: Observer. Calculate the hyper-rectangle that includes the point. |
| | | **-hrfmp()**: Observer. Return the hyper-rectangle that is built from the point. |
| | | **-display()**,<<(): Observers. HR display. |

Figure B.17: TDSO implementation of the **HR** class. (Continuation)

| Mar.97 | **Implementation 19 Class LeafEntry** Classes: MPoint, Cardinal, Char, Integer, HR, Interval, ostream | |
|---|---|---|
| 1 | *Superclasses:* | **-mp**: Multi-dimensional point. |
| 2 | *Structure:* | **-bn**: Blob number. |
| | mp: MPoint | **-LeafEntry()**: Constructors. |
| | bn:Cardinal | **-Mp(), Bn()**: Observers. Return the values |
| 3 | *Functions:* | of the leaf entry attributes. |
| | LeafEntry():LeafEntry | -=(): Mutator. Assignment. |
| | LeafEntry(MPoint, Cardinal):LeafEntry | -==(): Observer. Return one if |
| | LeafEntry(LeafEntry):LeafEntry | both leaf entries are equal in value. |
| | LeafEntry(Char):LeafEntry | -!=(): Observer. Return one if |
| | Mp():MPoint | both leaf entries are different in value. |
| | Bn():Cardinal | **-disjoint()**: Observer. Disjointness. |
| | =(LeafEntry): LeafEntry | **-match()**: Observer. Equality. |
| | ==(LeafEntry): Integer | **-inside()**: Observer. Multi-dimensional point |
| | !=(LeafEntry): Integer | is inside of the hyper-rectangle. |
| | disjoint(HR): Cardinal | **-coveredBy()**: Observer. Coverage. |

Figure B.18: TDSO implementation of the **LeafEntry** class.

| Mar.97 | |
|---|---|
| **Implementation {19}Class LeafEntry** | |
| match(MPoint):Cardinal | **-overlap()**: Observer. Overlapping. |
| inside(HR): Cardinal | **-slDisjoint()**: Observer. Spatial disjointness. |
| coveredBy(HR): Cardinal | **-slMatch()**: Observer. Spatial matching. |
| overlap(HR): Cardinal | **-slCoverg()**: Observer. General spatial |
| slDisjoint(HR): Cardinal | coverage. |
| slMatch(MPoint): Cardinal | **-slInside()**: Observer. Spatial inside. |
| slCoverg(HR): Cardinal | **-slOverlap()**: Observer. Spatial overlapping. |
| slInside(HR): Cardinal | **-slCover()**: Observer. Spatial coverage. |
| slOverlap(HR): Cardinal | **-overlapD()**: Observer. Overlapping in 1D. |
| slCover(HR): Cardinal | **-leftD()**: Observer. On the left in 1D. |
| overlapD(HR, Cardinal): Cardinal | **-rightD()**: Observer. On the right in 1D. |
| leftD(HR, Cardinal): Cardinal | **-leftOverlap()**: Observer. |
| rightD(HR, Cardinal): Cardinal | Overlapping on the left. |
| leftOverlap(HR, Cardinal): Cardinal | **-rightOverlap()**: Observer. |
| rightOverlap(HR, Cardinal): Cardinal | Overlapping on the right. |
| tMeet(HR, Cardinal):Cardinal | **-tMeet()**: Observer. Temporal meeting. |
| equal(MPoint, Cardinal):Cardinal | **-equal()**: Observer. Temporal equality. |
| start(HR, Cardinal):Cardinal | **-start()**: Observer. Time start equality. |
| finish(HR, Cardinal):Cardinal | **-finish()**: Observer. Time finish equality. |
| adjacent(HR, Cardinal):Cardinal | **-adjacent()**: Observer. Temporal adjacency. |
| precede(HR, Cardinal):Cardinal | **-precede()**: Observer. Temporal precedence. |
| follow(HR, Cardinal):Cardinal | **-follow()**: Observer. Temporal consequence. |
| during(HR, Cardinal):Cardinal | **-during()**: Observer. Temporal inclusion. |
| during(Interval):Cardinal | **-equal()**: Observer. Multi-dimensional |
| equal(MPoint):Cardinal | point equality. |
| Mp(MPoint) | **-Mp(), Bn()**: Mutators. Change the values |
| Bn(Cardinal) | of the leaf entry attributes. |
| display() | **-display(),<<()**: Observers. Leaf |
| <<(ostream, LeafEntry): ostream | entry display. |

Figure B.19: TDSO implementation of the **LeafEntry** class. (Continuation)

| Apr.96 | **Implementation {17}Class MRTree**<br>Classes: Character, Float, Integer, Cardinal, EleType, ListOf,<br>MPoint, Interval, HR, Entry, LeafEntry, RTreeNode | |
|---|---|---|
| 1 | *Superclasses:* | **-dim,height**: Dimension and actual height. |
| 2 | *Structure:*<br>dim,height,entLength,<br>leafEntLen,xDim,timeDim,<br>nTime,nCoord:Character<br>entNum,blobRoot,blobNum,<br>leafNum,bnf,blobSize,<br>insNum,delNum,diQNum,<br>nDQNum,maQNum,inQNum,<br>coQNum,slDiQNum,slMaQNum,<br>slInQNum,slCoQNum,<br>nDDQNum,lDQNum,rDQNum,<br>tmQNum,eqQNum,stQNum,<br>fiQNum,adQNum,preQNum,<br>foQNum,duQNum:Cardinal<br>emptyPer,deltaTime,delta,<br>tInsTime,tDelTime,tDiQTime,<br>tNDQTime,tMaQTime,tInQTime,<br>tCoQTime,tSlDiQTime,<br>tSlMaQTime,tSlInQTime,<br>tSlCoQTime,tNDDQTime,<br>tLDQTime,tRDQTime,tTmQTime,<br>tEqQTime,tStQTime,tFiQTime,<br>tAdQTime,tPreQTime,<br>tFoQTime,tDuQTime: Float<br>root: RTreeNode | **-entLength,leafEntLen**: Length of index and leaf entries, respectively.<br>**-xDim**: X coordinate position.<br>**-timeDim**: Time position.<br>**-nTime**: Number of time dimensions.<br>**-nCoord**: Number of coordinates.<br>**-entNum**: Current entries number.<br>**-blobRoot**: Blob root number.<br>**-blobNum,leafNum,bnf**: Actual number of index, leaf blobs in use, and assigned blobs.<br>**-blobSize**: Current blob size.<br>**-insNum**: Actual number of insertions.<br>**-delNum**: Actual number of deletions.<br>**-diQNum,nDQNum,maQNum,inQNum, coQNum**:Actual number of disjoint, not disjoint, exact match, inside, and cover dim-dimensional queries.<br>**-slDiQNum,slMaQNum,slInQNum, slCoQNum**: Current number of disjoint, exact match, inside, and cover queries (Spatial).<br>**-nDDQNum,lDQNum,rDQNum**: Actual number of not disjoint in dimension, left in dimension, right in dimension queries (1D). |
| 3 | *Functions:*<br>MRTree():MRTree<br>MRTree(Cardinal):<br>MRTree<br>∼MRTree()<br>Dim(),Height(),EntLength(),<br>LeafEntNum(),XDim(),TimeDim(),<br>NTime(),NCoord(): Character<br>EntNum(),BlobNum(),LeafNum(),<br>Bnf(),BlobSize(),InsNum(),<br>DelNum(),DiQNum(),NDQNum(),<br>MaQNum(),InQNum(),CoQNum(),<br>SlDiQNum(),SlMaQNum(), | **-tmQNum,eqQNum,stQNum,fiQNum, adQNum,preQNum,foQNum,duQNum**: Current number of temporal meeting, equal, start, finish, adjacent, precede, follow, and during temporal queries.<br>**-emptyPer,deltaTime,delta**: Actual percentage of empty space within nodes, unit of time to be used in temporal queries, and unit of space used in near/far queries.<br>**-tInsTime,tDelTime,tDiQTime,** |

Figure B.20: TDSO implementation of the **MRTree** class.

| Apr.96 | **Implementation 17 Class MRTree** | |
|---|---|---|
| | SlInQNum(),SlCoQNum(), | **tNDQTime,tMaQTime,tInQTime,** |
| | NDDQNum(),LDQNum(), | **tCoQTime**:Current cumulative |
| | RDQNum(),TmQNum(),EqQNum(), | insertion, deletion, disjoint, not |
| | StQNum(),FiQNum(),AdQNum(), | disjoint, exact match, inside, and |
| | PreQNum(),FoQNum(), | cover query time. |
| | DuQNum(): Cardinal | **-tSlDiQTime,tSlMaQTime,tSlInQTime,** |
| | isMultidim():Integer | **tSlCoQTime**: Current cumulative |
| | EmptyPer(),DeltaTime(), | spatial disjoint, exact match, |
| | Delta(),avgInsTime(), | inside, and cover query time. |
| | avgDelTime(),avgDiQtime(), | **-tNDDQTime,tLDQTime,tRDQTime**: |
| | avgNDQTime(),avgMaQTime | Current cumulative 1-dimensional |
| | avgInQTime(),avgCoQTime(), | not disjoint, left, and right in |
| | avgSlDQTime(),avgSlMaQTime(), | dimension query time. |
| | avgSlInQTime(), | **-tTmQTime,tEqQTime,tStQTime,** |
| | avgSlCoQTime(), | **tFiQTime,tAdQTime,tPreQTime,** |
| | avgNDDQTime(),avgLDQTime(), | **tFoQTime,tDuQTime**: Current |
| | avgRDQTime(),avgTmQTime(), | cumulative temporal meeting, equal, |
| | avgEqQTime(),avgStQTime(), | start, finish, adjacent, precede, |
| | avgFiQTime(),avgAdQTime(), | follow, and during query time. |
| | avgPreQTime(),avgFoQTime | **-root**: The R_tree root in main memory. |
| | avgDuQTime():Float | **-MRTree()**: Constructors. |
| | disjoint(HR,ListOf[Cardinal], | **-∼MRTree()**: Destructor. |
| | Cardinal,Cardinal):Float | **-Dim(),Height(),EntLength(),** |
| | nDisjoint(HR,ListOf[Cardinal], | **LeafEntLen(),XDim(),TimeDim(),** |
| | Cardinal,Cardinal):Float | **NTime(),NCoord(),EntNum(),BlobNum(),** |
| | match(MPoint,ListOf[Cardinal], | **LeafNum(),Bnf(),BlobSize(),** |
| | Cardinal,Cardinal):Float | **InsNum(),DelNum(),DiQNum(),** |
| | inside(HR,ListOf[Cardinal], | **NDQNum(),MaQNum(),InQNum(),** |
| | Cardinal,Cardinal):Float | **CoQNum(),SlDiQNum(),SlMaQNum(),** |
| | cover(HR,ListOf[Cardinal], | **SlInQNum(),SlCoQNum(),NDDQNum(),** |
| | Cardinal,Cardinal):Float | **LDQNum(),RDQNum(),TmQNum(),** |
| | slDisjoint(HR,ListOf[Cardinal], | **EqQNum(),StQNum(),FiQNum(),** |
| | Cardinal,Cardinal):Float | **AdQNum(),PreQNum(),FoQNum(),** |
| | slMatch(HR,ListOf[Cardinal], | **DuQNum()**: Observers. Each function |
| | Cardinal,Cardinal):Float | returns the current attribute's value. |
| | slInside(HR,ListOf[Cardinal], | **-isMultidim()**: Observer. Return 1 |
| | Cardinal,Cardinal):Float | if the R_tree has many dimensions. |
| | slCover(HR,ListOf[Cardinal], | **-EmptyPer(),DeltaTime(),Delta(),** |
| | Cardinal,Cardinal):Float | **avgInsTime(),avgDelTime(),** |
| | nDisjointD(HR,ListOf[Cardinal], | **avgDiQtime(),avgNDQTime(),** |
| | Cardinal,Cardinal,Cardinal):Float | **avgMaQTime(),avgInQTime(),** |

Figure B.21: TDSO implementation of the **MRTree** class. (Continuation)

| Apr.96 | |
|---|---|
| **Implementation {17}Class MRTree** | |
| leftD(HR,ListOf[Cardinal],Cardinal, Cardinal,Cardinal):Float<br>rightD(HR,ListOf[Cardinal], Cardinal,Cardinal,Cardinal):Float<br>tMeet(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>equal(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>start(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>finish(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>adjacent(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>precede(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>follow(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>during(Interval,ListOf[Cardinal], Cardinal,Cardinal):Float<br>insertEnt(LeafEntry,Float, Cardinal):MRTree<br>deletEnt(LeafEntry,Float, Cardinal):MRTree<br>cleanRTree(Cardinal,Cardinal): MRTree<br>chooseLeaf(RTreeNode,Entry, ListOf[EleType],Cardinal)<br>linearPickSeeds(RTreeNode,Cardinal)<br>splitNode(RTreeNode,RTreeNode)<br>adjustTree(RTreeNode,RTreeNode, ListOf[EleType],Cardinal)<br>findLeaf(RTreeNode,RTreeNode,Entry, Cardinal,ListOf[EleType],Cardinal)<br>condenseTree(RTreeNode, ListOf[EleType],Cardinal)<br>redistribution(RTreeNode,RTreeNode, RTreeNode,RTreeNode,Integer) | **avgCoQTime(),avgSlDQTime(), avgSlMaQTime(),avgSlInQTime(), avgSlCoQTime(),avgNDDQTime(), avgLDQTime(),avgRDQTime(), avgTmQTime(),avgEqQTime(), avgStQTime(),avgFiQTime(), avgAdQTime(),avgPreQTime(), avgFoQTime(),avgDuQTime()**: Observers. Each function returns the corresponding attribute value.<br>**-disjoint(),nDisjoint(),match(), inside(),cover(),slDisjoint(),slMatch(), slInside(),slCover(),nDisjointD(), leftD(),rightD(),tMeet(),equal(), start(),finish(),adjacent(), precede(),follow(),during()**: Observers. Each function executes a named query returning the number of disk accesses, number of retrieved objects, and time<br>**-insertEnt()**: Mutator. Insert a new entry returning the time used.<br>**-deletEnt()**: Mutator. Delete an existing entry returning the time used.<br>**-cleanRTree()**: Mutator. Delete all of the nodes of the R_tree.<br>**-chooseLeaf()**: Observer. Choose a leaf to store the new entry.<br>**-linearPickSeeds()**: Observer. Pick up the splitting point of a node.<br>**-splitNode()**: Mutator. Split a node following the splitting policy.<br>**-adjustTree()**: Mutator. Change the hyper-rectangle of the needed entries.<br>**-findLeaf()**: Observer. Find the leaf where is the entry that will be deleted.<br>**-condenseTree()**: Mutator. Change rectangles and delete nodes if needed.<br>**-hyperRecFromLeaves()**: Observer. |

Figure B.22: TDSO implementation of the **MRTree** class. (Continuation)

| Apr.96 | **Implementation {17}Class MRTree** | |
|---|---|---|
| | redistribution2(RTreeNode,RTreeNode, RTreeNode,Integer,Cardinal) | Prepare the hyper-rectangle of a leaf node. |
| | deletion(RTreeNode,RTreeNode, RTreeNode,RTreeNode,Integer, Cardinal) | **-overflow()**: Mutator. Decide over to redistribute or split an overflow node. |
| | deletion2(RTreeNode,RTreeNode, RTreeNode,Integer,Cardinal, Cardinal) | **-redistribution()**: Mutator. Redistribute entries among three nodes. |
| | <<(ostream,RTree) | **-redistribution2()**: Mutator. Redistribute entries among two nodes. |
| | displayNode(Cardinal) | **-deletion()**: Mutator. Delete a node when it has two brother nodes. |
| | hyperRecFromLeaves(RTreeNode,HR, HR,Integer):Integer | **-deletion2()**: Mutator. Delete a node when it has only one brother node. |
| | overflow(RTreeNode,ListOf[EleType]) | -<<(): Observer. Display on screen an MRTree object. |
| | | **-displayNode()**: Observer. Display on screen the demanded node. |

Figure B.23: TDSO implementation of the **MRTree** class. (Continuation)

# Appendix C

# Test functions and results.

---

This appendix presents the TDSO implementations of the test functions called by the test program mentioned in chapter 5, the obtained results after running the R_tree extensions, and the polygons of West Yorkshire.

## C.1 Test functions

The rest of the functions used in the test program are not included here as TDSO implementations. These functions are implemented in the code of each extension.

## C.2 Results

The results are shown in tables. Tables contain R_tree parameters, such as: blob size(BS), R_tree height(H), total number of used blobs(NB), total number of entries(NE), percentage of empty space within blobs(ES), average insertion time(AIT), average exact match searching time(AMST), average range searching time(ARST), average number of disk accesses during an insertion(IDA), average number of disk accesses during an exact match search (MDA), average number of disk accesses during a range search(RDA), and average number of retrieved objects during a range search(RRO). Additionally, results for extensions 3 and 4 include more parameters because range searches are particularised by spatio-temporal operators described in chapter 4. These new parameters are average disjoint time(ADT), average non disjoint time(A!DT), average inside or contain time(AICT), average cover or cov-

| Jun.96<br>**Implementation choices(from, ent, fd, op, top, prox, dop, tep, recl:Integer, hr1:HR, file:String)**<br>Classes: Integer, String, HR | | |
|---|---|---|
| 1 | case op =<br>1 createRTree(ent, recl, file)<br>2 insertRTree(hr1, recl, file)<br>3 deleteRTree(from, ent, recl, file)<br>4 if(top∈{1..8} ∧ prox∈<br>....{1, 2, 3}) then spatialQuery(fd, from,<br>..... ent, top, prox, recl, file)<br>... endif<br>... if(dop∈{1..4}) then unoDQuery(<br>..... from, ent, dim, dop, recl, file)<br>... endif<br>... if(dop∈{5..12}) then<br>.....double1DQuery(from, ent, dim, dim1,<br>..... dop, recl, file)<br>... endif<br>... if(tep∈{1, 2, 4}) then<br>.....unoDQuery(from, ent, dim, tep,<br>..... recl, file)<br>... endif<br>... if(tep∈{3, 5..11}) then<br>.....temporalQuery(from, ent, tep,<br>..... recl, file)<br>... endif<br>... if(tep = 12) then<br>.....double1DQuery(from, ent, dim,<br>..... dim1, tep, recl, file)<br>... endif<br>5 rt.display()<br>6 if(fd) then<br>..... [rt.displayNode(i)] i=1, rt.Bnf()<br>. .else [i=Read from the keyboard<br>..... if(i>0 ∧ i≤rt.Bnf()) then<br>..... rt.displayNode(i)<br>..... endif<br>..... yes=Read from the keyboard]<br>..... (yes≠1)<br>... endif<br>endcase | **-from, ent, fd, op, top, prox, dop, tep, recl, hr1, file**: Already defined in testProgram.<br>**-dim, dim1**: Integer: Dimension indicator.<br>**-rt**: RTree: An RTree instance.<br>**-yes**: Stop displaying.<br>**-createRTree()**: Create the desired RTree.<br>**-insertRTree()**: Insert a new entry in the RTree already created.<br>**-deleteRTree()**: Delete one o more entries of the RTree.<br>**-spatialQuery()**: Search in the RTree the blob number of the corresponding entries following top or prox operators.<br>**-unoDQuery()**: Search all the blob numbers of the corresponding entries following dim and dop or tep parameters.<br>**-double1DQuery()**: Search all the blob numbers of the corresponding entries following dim, dim1, and dop or tep parameters.<br>**-temporalQuery()**: Search all the blob numbers of the corresponding entries following the tep parameter.<br>**-display(), displayNode(), Bnf()**: RTree operations already defined in the RTree class. |

Figure C.1: TDSO implementation of **choices** function.

| Jun.96 **Implementation choices(from, ent, fd, op, top, prox, dop, tep, recl: Integer, hr1:HR, file:String)** | | |
|---|---|---|
| 1 | op=1 | RTree creation. |
| 2 | op=2 | Entry insertion. |
| 3 | op=3 | Entry deletions |
| 4 | op=4 | RTree search. |
| 5 | op=5 | Display RTree instance. |
| 6 | op=6 | Diaplay one or more RTree nodes. |

Figure C.2: TDSO implementation of **choices** function. Test cases.

| Jun.96 **Implementation createRTree(ent, recl:Integer, file:String)** Classes: Integer, String | | |
|---|---|---|
| 1 | [rec=record j of the file<br>. e=Entry(rec)<br>. rt.insertEnt(e, tim(j), da)<br>. dac=dac+da ] (j=1, ent) | **-rec**: String: File record that contain an entry.<br>**-j**: Integer: Counter.<br>**-rt**: RTree: An RTree instance. |
| 2 | if(ent>0) then<br>..... display dac, dac/ent<br>endif | **-e**: Entry: RTree entry containing the record.<br>**-tim**: Array[ent]Of Real: Time container.<br>**-da, dac**: Integer: Number of disk accesses and accumulate disk accesses, respectively.<br>**-Entry()**: Entry constructor.<br>**-insertEnt()**: RTree operation. |
| 1 | ent=0, da=0, dac=0 | No RTree creation. |
| 2 | ent=1, da=1, dac=1 | Create the RTree with one entry. |

Figure C.3: TDSO implementation of **createRTree** function.

| Jun.96 **Implementation insertRTree(ahr:HR, recl:Integer, file:String)** Classes: HR, Integer, String | | |
|---|---|---|
| 1 | p=position of the end-of-file | **-p**: Integer: Total number of bytes on the file. |
| 2 | a=p/recl | **-a**: Integer: Number of the actual last blob. |
| 3 | e.Bn(a) | **-e**: Entry: RTree new entry. |
| 4 | e.Hr(ahr) | **-rt**: RTree: An RTree instance. |
| 5 | rt.insertEnt(e, tim, da) | **-tim**: Real: Time container. |
| 6 | dac=dac+da | **-da, dac**: Integer: Number of disk accesses and |
| 7 | write record p on the file | accumulate disk accesses, respectively. |
| 8 | display dac, da | **-Bn(), Hr()**: Entry operations.<br>**-insertEnt()**: RTree operation. |
| 1 | ahr=NullHR, da=0, dac=0 | No RTree insertion. |
| 2 | ahr=ValidHR, da=1, dac=1 | Insert the new RTree entry. |

Figure C.4: TDSO implementation of **insertRTree** function.

| Jun.96<br>**Implementation deleteRTree(from, ent, recl:Integer, file:String)**<br>Classes: Integer, String | | |
|---|---|---|
| 1 | j=from | -**j**: Integer: Counter. |
| 2 | (j<(from+ent)) [rec=record j of the file<br>..... e=Entry(rec)<br>..... rt.deletEnt(e, tim(j), da)<br>..... dac, j=dac+da, j+1<br>..... ] | -**rec**: String: File record containing an entry.<br>-**e**: Entry: RTree entry.<br>-**rt**: RTree: An RTree instance.<br>-**tim**: Array[ent-from]Of Real: Time container. |
| 3 | if(ent>0) then<br>.... display dac, da/ent<br>endif | -**da, dac**: Integer: Number of disk accesses and accumulate disk accesses, respectively.<br>-**Entry()**: Entry constructor.<br>-**deletEnt()**: RTree operation. |
| 1 | from=0, ent=0, da=0, dac=0 | No RTree deletion. |
| 2 | from=0, ent=1, da=1, dac=1 | Delete one RTree entry. |

Figure C.5: TDSO implementation of **deletRTree** function.

ered by time(ACT), average non disjoint in one dimension time(A!D1T), average on the left in one dimension time(AL1T), average on the right in one dimension time(AR1T), average number of disk accesses during a disjoint search(DDA), average number of disk accesses during a non disjoint search(!DDA), average number of disk accesses during an inside or contain search(ICDA), average number of disk accesses during a cover or covered by search(CDA), average number of disk accesses during a non disjoint in one dimension search(!D1DA), average number of disk accesses during a left in one dimension search(L1DA), average number of disk accesses during a right in one dimension search(R1DA), average number of retrieved objects during a disjoint search(DRO), average number of retrieved objects during a non disjoint search(!DRO), average number of retrieved objects during an inside or contain search(ICRO), average number of retrieved objects during a cover or covered by search(CRO), average number of retrieved objects during a non disjoint in one dimension search(!D1RO), average number of retrieved objects during a left in one dimension search(L1RO), and average number of retrieved objects during a right in one dimension search(R1RO).

The average number of retrieved objects during a match search is not included in tables, because it is always equal to one. The others not included parameters are DRO, and CRO, because they are always equal to Database size - 1, and zero, respectively.

| Data-base size | BS | Ex-ten-sion | H | NB | NE | ES | AIT [ms] | AMST [ms] | ARST [ms] | IDA | MDA and RDA | RRO |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1000 | 512 | 1 | 2 | 57 | 1056 | 25.90 | 2.00 | 10.00 | 8.00 | 4 | 18 | 16 |
| | | 2 | 2 | 48 | 1047 | 12.75 | 3.00 | 17.00 | 8.00 | 4 | 25 | 16 |
| | 1K | 1 | 1 | 26 | 1025 | 21.15 | 3.00 | 14.00 | 8.00 | 3 | 14 | 16 |
| | | 2 | 1 | 22 | 1021 | 7.18 | 3.00 | 12.00 | 8.00 | 3 | 12 | 16 |
| | 2K | 1 | 1 | 13 | 1012 | 22.93 | 4.00 | 11.00 | 8.00 | 2 | 6 | 16 |
| | | 2 | 1 | 12 | 1011 | 16.58 | 5.00 | 16.00 | 8.00 | 2 | 100 | 16 |
| 2770 | 512 | 1 | 2 | 157 | 2926 | 25.45 | 3.61 | 31.77 | 11.55 | 5 | 53 | 43 |
| | | 2 | 2 | 133 | 2902 | 12.72 | 3.61 | 44.04 | 46.21 | 5 | 70 | 43 |
| | 1K | 1 | 2 | 75 | 2844 | 24.16 | 3.61 | 32.85 | 11.55 | 3 | 35 | 43 |
| | | 2 | 2 | 64 | 2833 | 11.47 | 3.97 | 37.19 | 11.55 | 3 | 39 | 43 |
| | 2K | 1 | 1 | 37 | 2806 | 24.91 | 5.05 | 27.44 | 11.55 | 2 | 19 | 43 |
| | | 2 | 1 | 31 | 2800 | 10.57 | 5.42 | 44.40 | 11.55 | 3 | 26 | 43 |
| 5000 | 512 | 1 | 2 | 277 | 5276 | 23.81 | 3.60 | 57.00 | 25.60 | 5 | 93 | 79 |
| | | 2 | 2 | 241 | 5240 | 13.03 | 3.80 | 80.20 | 25.60 | 5 | 126 | 79 |
| | 1K | 1 | 2 | 136 | 5135 | 24.49 | 4.20 | 57.20 | 25.60 | 4 | 64 | 79 |
| | | 2 | 2 | 113 | 5112 | 9.52 | 4.60 | 75.40 | 25.60 | 4 | 77 | 79 |
| | 2K | 1 | 1 | 65 | 5064 | 22.86 | 5.40 | 47.80 | 25.60 | 3 | 31 | 79 |
| | | 2 | 1 | 54 | 5053 | 7.35 | 5.60 | 75.60 | 25.60 | 3 | 44 | 79 |
| 10000 | 512 | 1 | 3 | 560 | 10559 | 24.58 | 4.10 | 114.00 | 51.20 | 5 | 188 | 157 |
| | | 2 | 3 | 483 | 10482 | 13.19 | 4.30 | 170.40 | 51.20 | 5 | 255 | 158 |
| | 1K | 1 | 2 | 263 | 10262 | 21.96 | 4.80 | 101.70 | 51.20 | 4 | 111 | 158 |
| | | 2 | 2 | 226 | 10225 | 9.51 | 4.60 | 141.40 | 51.20 | 4 | 145 | 158 |
| | 2K | 1 | 2 | 132 | 10131 | 24.01 | 6.00 | 94.70 | 51.20 | 3 | 63 | 158 |
| | | 2 | 2 | 107 | 10106 | 6.49 | 5.80 | 146.00 | 51.20 | 3 | 87 | 158 |

Table C.1: Results for objects uniformly distributed in a space of two dimensions. Extensions 1 and 2.

Figures C.6, and C.7 correspond to average *inside* or *contain* time (AICT), and average *cover* or *covered by* time (ACT).

Figures C.8, C.9 and C.10 show disjoint, left and right in one dimension hit ratio curves, respectively.

Not disjoint, inside or contain, and not disjoint in one dimension hit ratio curves are presented in figures C.11, C.12, and C.13, respectively.

## C.3 Polygons of West Yorkshire

We include an example of the obtained data for the polygons of West Yorkshire. Similar information was obtained for the polygons of England counties, which are not included because of lack of space.

SEGMENTS

Z03BPAP Z08CZAU 39951 41301 6 -6 10 -13 6 -8 5 -5 4 -5 5 -5 8 -8 7 -6 7 -7 3 -4 6 -4 1 -2 4 3 3 3 2 3 2 6 1 3 1 3 3 2 3 2 0 0 4 -5 2 -4 3 -4 5 -6 3 -4 -6 -5 -5 -5 -4 -5 -4 -4 -3 -1 3 -7 6 -10 5 -10 3 -7 4 -6 5 -6 4 -9 6 -11 6 -12 5 -9 7 -15 5 -8 5 -11 5 -11 5 -9 4 -8 4 -6 4 -7 2 -4 4 -6 2 -3 4 -4 5 -6 5 -5 4 -5 4 -4 4 -4 3 -4 6 -5 8 -5 6 -4 7 -4 4 -2 2 -2 4 -3 3 -3 5 -7 2 -3 2 -4 /

Z03BQAJ Z08CYAM 39673 42034 -6 -6 -6 -5 -5 -5 -2 -5 -3 -6 -1 -5 -2 -7 1 -9 0 -5 1 -12 1 -4 2 -11 2 -9 2 -10 3 -6 2 -6 2 -2 6 -3 6 -3 4 -3 5 -4 4 -5 4 -8 4 -4 3 -3 5 -3 4 -3 7 -7 6 -6 6 -4 6 -5 7 -3 4 -3 6 -4 3 -3 3 -3 7 -10 5 -10 1 -7 2 -8 1 -7 0 -7 0 -8 -1 -7 -1 -7 -1 -4 -3 -7 -3 -7 -6 -10 -6 -10 -2 -3 -5 -9 -1 -2 -1 -3 -1 -5 1 -5 0 -4 2 -3 2 -4 0 -2 -1 -3 -1 -1 -3 -3 -4 0 1 -4 0 -5 -3 -13 -2 -8 -1 -9 -1 -7 1 -7 0 -5 1 -9 2 -7 3 -7 2 -5 3 -6 5 -5 6 -7 8 -6 8 -7 3 -2 6 -3 5 -3 6 -2 6 -2 5 -2 6 -3 3 -3 1 -3 3 -7 0 -6 1 -8 1 -7 -1 -10 -1 -8 -1 -8 -2 -6 -3 -6 -2 -5 -1 -3 /

Z03BQAQ Z08CYAM 39806 41474 0 -6 1 -6 1 -5 5 -7 5 -9 5 -11 3 -7 4 -9 3 -9 2 -7 1 -7 1 -8 1 -7 1 -7 1 -5 1 -3 3 -6 1 -7 2 -6 6 -13 2 -5 1 -1 8 1 12 1 5 1 3 0 4 -1 7 -4 5 -2 4 -1 5 -1 2 1 9 3 9 5 5 3 6 2 2 1 /

Z03BQAQ Z08CZAU 39942 41332 8 -15 1 -3 4 -8 2 -3 -6 -2 /

| Database size | BS | Extension | H | NB | NE | ES | AIT [ms] | ADT [ms] | A!DT [ms] | AMST [ms] | AICT [ms] |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1000 | 512 | 3 | 2 | 57 | 1056 | 25.90 | 2.00 | 403.29 | 11.00 | 8.00 | 5.50 |
| | | 4 | 2 | 48 | 1047 | 12.75 | 2.00 | 283.00 | 16.00 | 12.00 | 16.00 |
| | 1K | 3 | 1 | 26 | 1025 | 21.15 | 2.00 | 340.00 | 14.00 | 11.00 | 13.99 |
| | | 4 | 1 | 22 | 1021 | 7.18 | 3.00 | 308.00 | 13.00 | 11.00 | 13.00 |
| | 2K | 3 | 1 | 14 | 1013 | 28.36 | 3.99 | 344.00 | 11.00 | 9.00 | 11.00 |
| | | 4 | 1 | 12 | 1011 | 16.58 | 4.00 | 303.00 | 18.00 | 16.00 | 17.00 |
| 2770 | 512 | 3 | 2 | 157 | 2926 | 25.45 | 2.53 | 3153.58 | 32.37 | 21.66 | 15.34 |
| | | 4 | 2 | 133 | 2902 | 12.72 | 2.89 | 1659.20 | 45.49 | 33.57 | 43.68 |
| | 1K | 3 | 2 | 75 | 2844 | 24.16 | 3.25 | 2148.74 | 34.30 | 24.91 | 32.85 |
| | | 4 | 3 | 64 | 2833 | 11.47 | 2.53 | 2388.81 | 40.07 | 32.13 | 38.99 |
| | 2K | 3 | 2 | 36 | 2805 | 22.86 | 4.33 | 3355.23 | 29.24 | 22.74 | 28.88 |
| | | 4 | 1 | 31 | 2800 | 10.57 | 4.70 | 4084.12 | 46.57 | 41.52 | 45.85 |
| 5000 | 512 | 3 | 2 | 227 | 5276 | 23.81 | 2.60 | 9777.40 | 58.60 | 37.40 | 54.80 |
| | | 4 | 2 | 241 | 5240 | 13.03 | 3.20 | 7788.40 | 82.00 | 59.40 | 78.00 |
| | 1K | 3 | 2 | 134 | 5133 | 23.39 | 3.60 | 8225.20 | 58.40 | 41.20 | 55.20 |
| | | 4 | 2 | 113 | 5112 | 9.52 | 3.80 | 5397.00 | 81.20 | 64.20 | 77.60 |
| | 2K | 3 | 1 | 65 | 5064 | 22.86 | 4.60 | 11580.60 | 54.40 | 40.40 | 51.60 |
| | | 4 | 1 | 54 | 5053 | 7.35 | 4.60 | 9620.00 | 80.80 | 70.40 | 79.00 |
| 10000 | 512 | 3 | 3 | 560 | 10559 | 24.58 | 3.10 | 72974.60 | 118.90 | 73.70 | 109.30 |
| | | 4 | 3 | 483 | 10482 | 13.19 | 2.87 | 62940.50 | 157.80 | 129.40 | 177.50 |
| | 1K | 3 | 2 | 263 | 10262 | 21.96 | 4.00 | 34272.50 | 115.70 | 74.70 | 102.50 |
| | | 4 | 2 | 226 | 10225 | 9.51 | 4.70 | 29450.00 | 153.80 | 137.20 | 163.80 |
| | 2K | 3 | 2 | 132 | 10131 | 24.01 | 4.80 | 17201.20 | 107.60 | 74.30 | 97.20 |
| | | 4 | 2 | 107 | 10106 | 6.49 | 5.10 | 13943.40 | 164.20 | 135.80 | 154.40 |

Table C.2: Results for objects uniformly distributed in a space of two dimensions. Extensions 3 and 4.

| Database size | BS | Extension | ACT [ms] | A!D1T [ms] | AL1T [ms] | AR1T [ms] | !DRO | ICRO | !D1RO | L1RO | R1RO |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1000 | 512 | 3 | 5.50 | 71.25 | 182.47 | 148.76 | 16 | 0 | 120 | 439 | 439 |
| | | 4 | 17.00 | 23.00 | 135.00 | 137.00 | | | | | |
| | 1K | 3 | 13.00 | 23.00 | 127.00 | 117.00 | | | | | |
| | | 4 | 14.00 | 20.00 | 123.00 | 116.00 | | | | | |
| | 2K | 3 | 11.00 | 19.00 | 94.99 | 123.00 | | | | | |
| | | 4 | 19.00 | 23.00 | 120.00 | 128.00 | | | | | |
| 2770 | 512 | 3 | 15.70 | 478.06 | 1664.12 | 1237.44 | 43 | 1 | 332 | 1218 | 1218 |
| | | 4 | 46.21 | 115.52 | 1389.53 | 1377.62 | | | | | |
| | 1K | 3 | 34.30 | 92.42 | 1020.58 | 1036.10 | | | | | |
| | | 4 | 41.16 | 85.20 | 984.84 | 981.23 | | | | | |
| | 2K | 3 | 31.05 | 68.95 | 608.66 | 889.90 | | | | | |
| | | 4 | 49.46 | 68.23 | 429.24 | 605.78 | | | | | |
| 5000 | 512 | 3 | 28.30 | 245.99 | 3587.80 | 3955.20 | 79 | 1 | 603 | 2198 | 2198 |
| | | 4 | 81.80 | 298.80 | 4709.80 | 4741.00 | | | | | |
| | 1K | 3 | 58.60 | 241.40 | 3518.60 | 3851.60 | | | | | |
| | | 4 | 82.60 | 261.20 | 3659.00 | 4423.40 | | | | | |
| | 2K | 3 | 55.40 | 178.20 | 2123.40 | 3238.20 | | | | | |
| | | 4 | 84.80 | 142.40 | 1288.60 | 2056.80 | | | | | |
| 10000 | 512 | 3 | 114.20 | 979.10 | 10159.60 | 10772.70 | 158 | 3 | 1200 | 4399 | 4399 |
| | | 4 | 197.40 | 997.30 | 10893.20 | 10920.10 | | | | | |
| | 1K | 3 | 108.50 | 912.00 | 16945.90 | 19584.30 | | | | | |
| | | 4 | 172.60 | 973.20 | 17282.30 | 21778.20 | | | | | |
| | 2K | 3 | 103.80 | 747.50 | 14960.80 | 15875.30 | | | | | |
| | | 4 | 165.90 | 654.90 | 12763.70 | 11459.20 | | | | | |

Table C.3: Results for objects uniformly distributed in a space of two dimensions. Extensions 3 and 4. (Continuation)

| Data-base size | BS | Ex-ten-sion | IDA | DDA | !DDA | MDA | ICDA | CDA | !D1DA | L1DA | R1DA |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1000 | 512 | 3 | 4 | 53 | 18 | 13 | 18 | 18 | 38 | 43 | 42 |
| | | 4 | | 44 | 25 | 20 | 25 | 25 | 30 | 39 | 37 |
| | 1K | 3 | 3 | 25 | 14 | 11 | 14 | 14 | 19 | 21 | 22 |
| | | 4 | | 21 | 12 | 10 | 12 | 12 | 14 | 17 | 18 |
| | 2K | 3 | 2 | 13 | 7 | 5 | 7 | 7 | 9 | 11 | 11 |
| | | 4 | | 11 | 10 | 9 | 10 | 10 | 10 | 10 | 10 |
| 2770 | 512 | 3 | 5 | 148 | 53 | 36 | 53 | 53 | 107 | 120 | 121 |
| | | 4 | | 124 | 70 | 54 | 70 | 70 | 85 | 111 | 105 |
| | 1K | 3 | 3 | 72 | 35 | 27 | 35 | 35 | 52 | 64 | 62 |
| | | 4 | | 61 | 39 | 32 | 39 | 39 | 43 | 54 | 52 |
| | 2K | 3 | 2 | 35 | 18 | 14 | 18 | 18 | 24 | 30 | 29 |
| | | 4 | 3 | 30 | 26 | 24 | 26 | 26 | 28 | 29 | 28 |
| 5000 | 512 | 3 | 5 | 262 | 93 | 64 | 93 | 93 | 152 | 214 | 214 |
| | | 4 | | 225 | 126 | 96 | 126 | 126 | 154 | 203 | 190 |
| | 1K | 3 | 4 | 129 | 59 | 44 | 59 | 59 | 89 | 112 | 109 |
| | | 4 | | 108 | 77 | 64 | 77 | 77 | 86 | 102 | 95 |
| | 2K | 3 | 3 | 64 | 33 | 25 | 33 | 33 | 44 | 54 | 54 |
| | | 4 | | 59 | 44 | 39 | 44 | 44 | 46 | 49 | 49 |
| 10000 | 512 | 3 | 5 | 346 | 188 | 126 | 188 | 188 | 274 | 434 | 437 |
| | | 4 | | 319 | 255 | 194 | 255 | 255 | 270 | 415 | 410 |
| | 1K | 3 | 4 | 254 | 111 | 80 | 111 | 111 | 176 | 221 | 217 |
| | | 4 | | 218 | 145 | 119 | 145 | 145 | 154 | 201 | 198 |
| | 2K | 3 | 3 | 154 | 63 | 48 | 63 | 63 | 89 | 110 | 110 |
| | | 4 | | 128 | 87 | 77 | 87 | 87 | 82 | 98 | 98 |

Table C.4: Results for objects uniformly distributed in a space of two dimensions. Extensions 3 and 4. (Continuation)

| Data-base size | H | NB | NE | ES | AIT [ms] | AMST [ms] | ARST [ms] | IDA | MDA and RDA | RRO |
|---|---|---|---|---|---|---|---|---|---|---|
| 1000 | 2 | 40 | 1039 | 27.85 | 3.00 | 14.00 | 8.00 | 3 | 14 | 1 |
| 2770 | | 106 | 2875 | 24.66 | 4.33 | 29.96 | 11.55 | 4 | 31 | 2 |
| 5000 | | 192 | 5191 | 24.90 | 5.00 | 56.00 | 25.60 | 4 | 59 | 3 |
| 10000 | | 374 | 10373 | 22.96 | 4.90 | 145.80 | 51.20 | 5 | 162 | 31 |

Table C.5: Results for objects uniformly distributed in two spatial dimensions and in one time dimension. Extension 1 (BS=1K)

| Ex-ten-sion | H | NB | NE | ES | AIT [ms] | A!DT [ms] | AMST [ms] | AICT [ms] | A!D1T [ms] | AL1T [ms] | AR1T [ms] |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 184 | 10183 | 23.14 | 5.80 | 163.80 | 136.40 | 160.70 | 480.20 | 13.68 | 13.89 |
| 4 | 2 | 152 | 10151 | 7.25 | 5.80 | 194.90 | 166.70 | 189.10 | 598.70 | 20.16 | 19.87 |

| | !DRO | ICRO | !D1RO | L1RO R1RO | IDA | !DDA | MDA | ICDA | !D1DA | L1DA | R1DA |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 87 | 5 | 84 | 45 | 4 | 10 | 91 | 10 | 14 | 16 | 16 |
| 4 | 87 | 5 | 84 | 45 | 4 | 11 | 99 | 11 | 12 | 14 | 13 |

Table C.6: Results for objects uniformly distributed in two spatial dimensions and in one time dimension. (BS=2K, Database size=10.000)

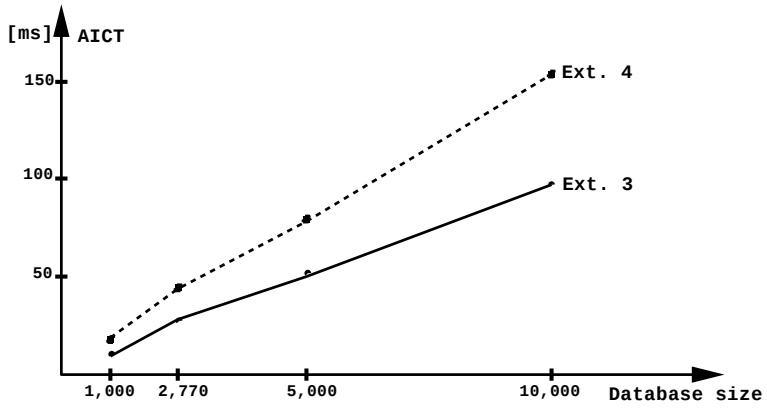



Figure C.6: AICT vs. data set size for extensions 3 and 4. (File: s2D.mbr, blob size=2KB)

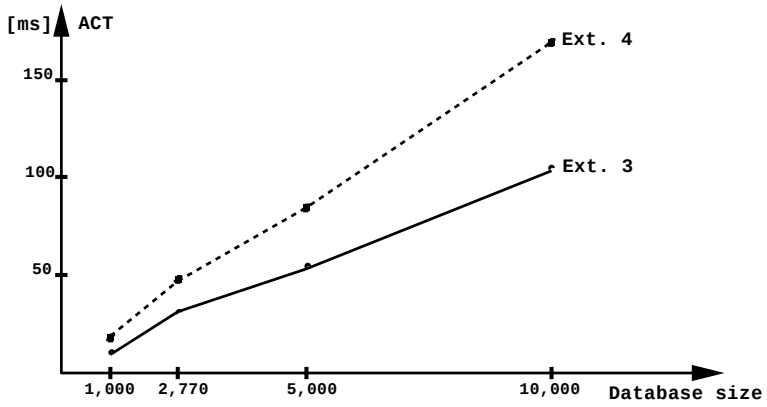



Figure C.7: ACT vs. data set size for extensions 3 and 4. (File: s2D.mbr, blob size=2KB)

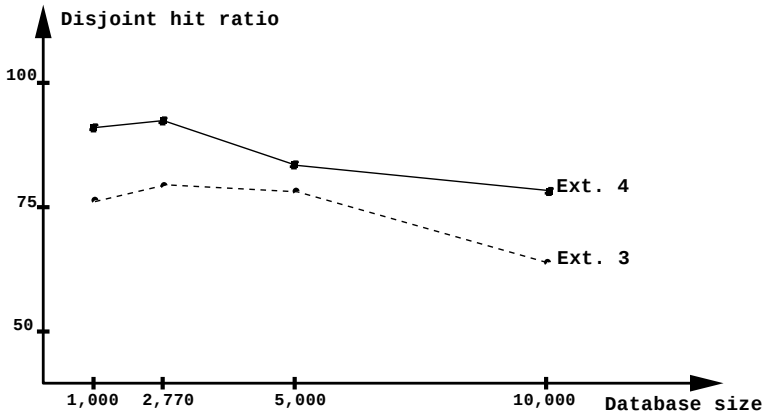



Figure C.8: The disjoint hit ratio vs. data set size for extensions 3 and 4. (File: s2D.mbr, blob size=2KB)

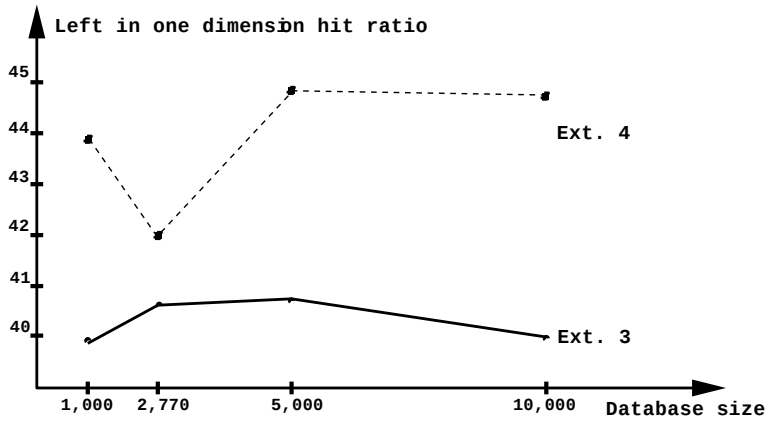



Figure C.9: The left in one dimension hit ratio vs. data set size for extensions 3 and 4. (File: s2D.mbr, blob size=2KB)

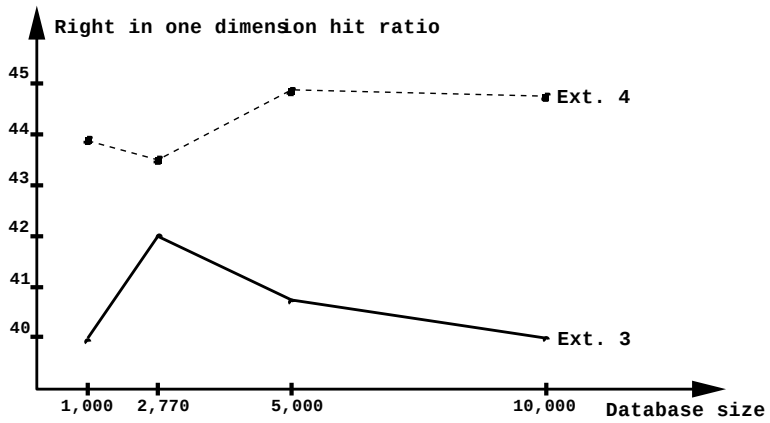



Figure C.10: The right in one dimension hit ratio vs. data set size for extensions 3 and 4. (File: s2D.mbr, blob size=2KB)

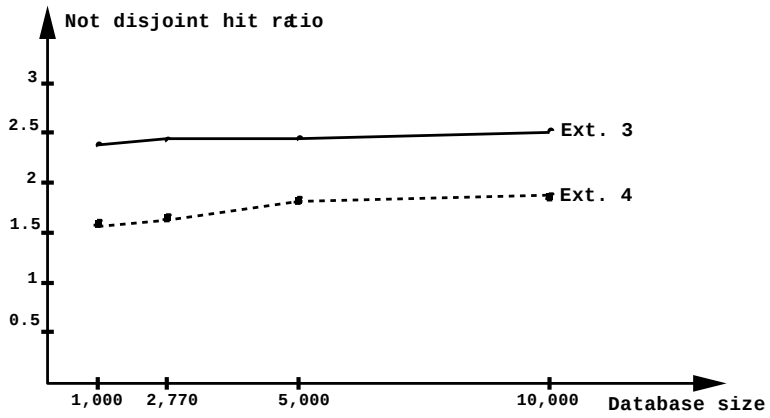



Figure C.11: The not disjoint hit ratio vs. data set size for extensions 3 and 4. (File: s2D.mbr, blob size=2KB)

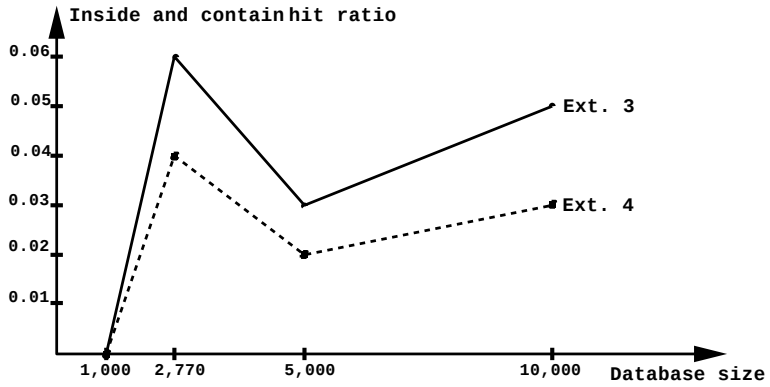



Figure C.12: The inside or contain hit ratio vs. data set size for extensions 3 and 4. (File: s2D.mbr, blob size=2KB)

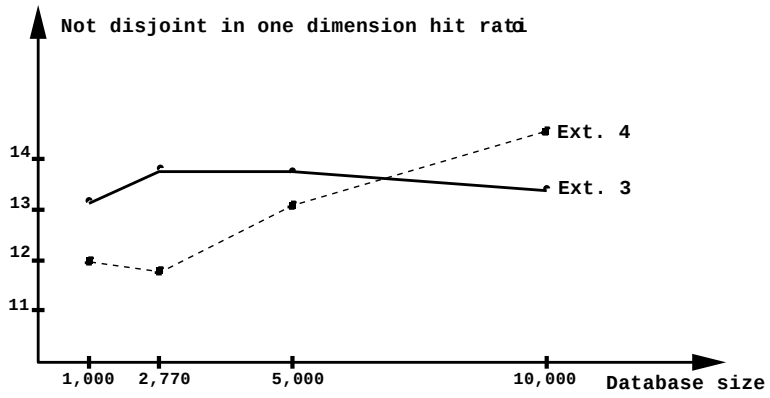



Figure C.13: The not disjoint in one dimension hit ratio vs. data set size for extensions 3 and 4. (File: s2D.mbr, blob size=2KB)

| xmin | xmax | ymin | ymax | Min(x+y) | Max(x+y) | Min(y-x) | Max(y-x) | Polygon name |
|---|---|---|---|---|---|---|---|---|
| 39850.00 | 41093.00 | 40631.00 | 41762.00 | 81115.00 | 82431.00 | -36.00 | 1466.00 | Z08CZAU |
| 40471.00 | 41969.00 | 40283.00 | 41015.00 | 81074.00 | 82767.00 | -1257.00 | 169.00 | Z08CZAY |
| 38959.00 | 40279.00 | 41927.00 | 43660.00 | 81085.00 | 83482.00 | 2060.00 | 4252.00 | Z08CYAB |
| 39429.00 | 40927.00 | 41222.00 | 42545.00 | 81180.00 | 83096.00 | 1147.00 | 2522.00 | Z08CYAM |

Table C.7: Example of the octagon points obtained and their associated polygon name.

Z05CEAP Z08DBAU 44570 40907 8 -3 9 -6 7 -1 3 -1 2 4 8 5 8 4 12 2 21 10 16 6 9 5 4 3 3 4 2 3 4 3 12 0 5 2 14 11 14 12 10 8 7 5 5 0 5 0 5 7 10 5 4 6 1 5 -1 4 2 3 4 1 2 0 0 7 -4 10 -5 8 9 13 8 11 6 -3 5 0 7 3 5 1 15 -1 22 0 11 0 4 3 2 6 5 4 5 4 4 -6 3 2 5 4 8 -1 6 3 6 2 8 8 4 6 -4 13 -1 10 2 3 9 5 12 8 8 8 -4 6 -9 -6 -2 -2 -9 11 -1 0 8 8 -2 4 -8 -3 -7 -4 3 10 4 6 2 3 14 4 19 6 9 4 3 -7 -10 -6 3 -6 14 10 8 4 -5 8 -2 5 -8 10 -2 3 -4 14 -6 19 -2 7 5 4 9 1 13 4 11 1 4 0 5 3 -3 10 -2 14 -4 8 0 6 -2 15 0 9 -2 16 -2 13 0 5 11 4 12 1 10 3 15 6 19 10 8 5 3 3 / END

After processing the file that contains records shown before, we obtain a file ready to be read for extension 6. A sample of this file in ASCII form is presented in table C.7. The first eight points conform the octagon corresponding to the polygon name presented at the final of the line.