Chapter 3

Access methods for similarity search

3.1 Introduction

Searching for data in large repository is intrinsically a difficult task. In order to perform search operations efficiently a physical level support is needed. Accordingly, several access methods were designed to optimize the execution of both exact match and similarity search operation.

In the following we introduce three classes of access methods: sequential scan techniques, hashing techniques, and tree-based techniques. Since the approaches proposed in this thesis are applied to tree-based access methods, we briefly introduce sequential scan and hashing techniques. Then, we dedicate more space to tree-based access methods. We define a general structure for tree-based access methods and we also introduce the basic similarity search algorithms. The definition of some specific tree based access methods is finally discussed distinguishing four different application areas: one dimensional data, multi dimensional data, spatial data, and metric data.

3.2 Access methods for similarity search

Literature offers several excellent surveys related to access methods for similarity search. As an example consider [GG98, BBK01, CNBYM01]. Here we describe some of the most significant approaches related to the issues discussed in this thesis.

In this chapter, for convenience, we broadly classify the access structures organizations for efficient processing of similarity search queries into three categories:

a) sequential scan

b) hashing

c) tree-based

Access structure organizations for *sequential scan* organize data in such a way that a sequential scan of the whole data base can be efficiently executed. *Hashing methods*, organize data in buckets according to some hashing functions so that only a few buckets should be accessed to process a query. *Tree-based access methods*, organize data in a tree hierarchy. Only a few nodes of the tree should be accessed to process a query.

Comparison of different access methods can be performed considering several aspects. However, as suggested in [GG98], among the important features of access methods is worth considering the followings:

Dynamicity: Access methods should allow insertion and removal of entries, without affecting their performance

Secondary storage: Since databases contains huge amount of data, data structures

used to organize entries cannot be maintained into main memory and should be designed to make an efficient use of secondary storage.

Independence of input sequence: Performance of access methods should not depend on the order in which data are inserted in the index.

Scalability: Performance of access methods should not degrade quickly as the size of the database increases.

Space efficiency: The space required to maintain the access structure should be limited.

The algorithms and the techniques presented in this thesis are mainly devoted to tree-based access method. Therefore, in the following we will briefly introduce sequential scan and hashing techniques, then we will dedicate more space to the tree-based techniques.

3.2.1 Sequential scan

The basic technique to process similarity search queries defined in Section 2.4 is to perform a single scan of the entire database. In order to perform this efficiently, data can be densely stored on contiguous blocks of a secondary storage. The *sequential scan* techniques, access sequentially all data of the database, according to the ordering used to place data on the secondary storage. This technique is faster than accessing randomly small blocks spread widely on the secondary storage. In fact, data stored contiguously can be transferred into main memory very fast, since no head seek latency occurs. On the other hand, in case of random block transfer, head seek time is is the main reason for inefficiency. Sequential scan algorithms transfer into main

memory large blocks of data, depending on the available amount of main memory. After a block is transferred, data contained in it are processed, then a new block is transferred, until all data are processed. Performance of sequential scan techniques is linearly proportional to the size of the database. Their aim is to improve performance of similarity search algorithms by minimizing the head seek overhead. However, they cannot reduce the data transfer and data process overheads, that are linear to the size of the database.

A relevant approach for data represented in vector spaces, based on the sequential scan technique, is the VA-File (Vector Approximation File) [WSB98]. It reduces the size of the data set using a coarse approximate representation of data objects. In this representation more objects may collapse in one coarse object. Sequential scan is performed on the reduced data set containing the coarse representation. The resulting coarse result set is used to lookup in the original data set to find the actual result set.

Notice that in some applications, head seek and data transfer overhead have a minor impact with respect to data process overhead. In fact, data process overhead is mainly due to distance computation between data objects and query objects and, some applications, as for instance genetic databases, require distance functions whose computation costs much more than disk accesses.

Hashing and tree-based access methods, on the other hand, use data structures that also aim at minimizing the amount of data transferred and processed.

3.2.2 Hashing

The idea behind the hashing techniques [Knu98, WFHC92] is that data objects should be allocated in a certain number of buckets in order to access just a few of them when

a query should be processed. Each bucket typically corresponds to a disk page and has a specific capacity. A hash function takes as input an object and returns the identifier of a bucket. The object is added to the obtained bucket. A typical technique used to handle the situation where adding a new object in a bucket would exceed the bucket's capacity is to store the additional entries in overflow buckets associated with the original bucket.

Hashing techniques for exact match search use arbitrary hash functions: when a query is processed, the hash function is applied to the query object. The exactly matching data object is searched in the obtained bucket and eventually in the corresponding overflow buckets.

Hashing techniques for similarity search, as for instance range queries, use hash functions that preserve the closeness of data objects. Close (similar) objects are stored in the same bucket.

A significant hashing approach for multi-dimensional data, therefore applicable to data represented in vector spaces and distances measured using functions of the Minkowski family, is the *grid-file* [NHS84]. It partitions the search space symmetrically in all dimensions. Specifically the grid-file put a d-dimensional orthogonal grid on top of the search space obtaining a set of hyper-rectangular cells, whose shape and size might be different, depending on the regularity of the grid itself. Each cell or a group of cells are associated with data buckets. Each cell is associated with one bucket but each bucket might contain several adjacent cells. When a new object (that is a vector) should be added to the data set, first the cell that contains it is identified. If the bucket containing the cell is not full, then the object is inserted. If the bucket is full, there are two possibilities. If the bucket contains several cells, one checks if

one of the existing hyperplane of the grid can be used for splitting the bucket. If there is not such an hyperplane or if the bucket is associated with only one cell, then a new hyperplane is added to the grid so that the bucket is split. Exact queries are executed by locating the cell that contains the query object and searching inside the corresponding bucket. Range queries are processed by locating cells that overlap the query region and searching in the corresponding buckets.

Another interesting hashing approach for similarity search in metric spaces is the *D-Index* [GSZ02], whose basic techniques were also proposed in [GSZ00, GSZ01]. It is a multilevel hash structure that takes advantage of the idea of the *excluded middle partitioning*, also exploited in [Yia99] for building an access method based on excluded middle vantage point. In each level of the hash structure a certain number of buckets are defined. Each level is associated with a hash function that, given an object, assigns it to either a specific bucket or no buckets in the corresponding level. Objects that are associated with no buckets in a level become candidate to be stored in the next level. It might happen that some objects are excluded to be stored in all levels. These remaining objects are stored in a separate *excluded bucket*. In this access structure, each object is stored in one bucket of the multilevel structure or in the excluded bucket. Queries can be processed, by accessing at most one bucket per level plus the excluded bucket. The upper bound on the number of disk access is limited by the number of levels, and the number of required distance computations can be significantly reduced by using some pre-computed distances. Distance computation between query objects and some accessed objects sometime is not performed since, by using pre-computed distances, it is possible to infer that they do not belong to the result set.

3.3 Tree-based access methods and similarity search algorithms

Approaches classified as tree-based access methods currently dominate the field. Their basic principle is the hierarchical decomposition of the data space. Techniques developed in this thesis suppose that tree-based access methods are used so we devote more space to their description.

All tree-based access methods have some common features, concerning their structure. All of them, for instance, hierarchically organize the space by using regions, however there are differences concerning the definition of the used regions and the way the trees are built and maintained. For instance, the Generalized search trees (GIST) [HNP95] subsumes most of the characteristics of tree-based access methods under a generic architecture, so it can be used as a reliable framework for the implementation of new access methods.

In the following we describe the general structure of tree-based access methods, we outline the generic similarity search algorithms that exploit this structure, finally we discuss some specific implementation of tree-based access methods.

3.3.1 General structure

Tree-based access methods partition objects and store elements (set of objects) of the partition in buckets, as hashing methods also do. However, while in hashing methods a bucket identifier is obtained by applying a hash function to data objects, in tree based access methods, buckets are accessed by traversing the tree structure.

The tree structure is composed of nodes that contain pointers to other nodes.

Nodes that do not refer other nodes are called *leaf nodes*. The remaining are called *internal nodes*. Each node of the tree is associated with a region that represents a subspace of the entire search space. The region associated with a node N_i is contained in the region associated with the N_i's parent node and contains all regions associated with the N_i's descendants.

A leaf node is simply a container of references to data records along with the objects that represent them. Let's suppose that N_i is a leaf node of a tree. The generic structure of N_i is the following:

$$N_i = \left((p_1^i, O_1^i), \ldots, (p_{n_i}^i, O_{n_i}^i)\right)$$

where n_i is the number of objects contained in N_i, p_j^i is a pointer to a record (for instance an image), and the object O_j^i is the corresponding representation (for instance the image color histogram). All objects $O_j^i, 1 \leq j \leq n_i$, belong to the region associated with N_i. Notice that in traditional database systems, objects O_j^i are typically called keys.

The definition of the region associated with a node is stored in its parent node, along with its reference. In fact, an internal node is a container of references to other nodes, either internal or leaf nodes, along with the definition of their associated region. Let's suppose that N_i is an internal node of the tree. The generic structure of N_i is the following:

$$N_i = \left((r_1^i, \mathcal{R}_1^i), \ldots, (r_{n_i}^i, \mathcal{R}_{n_i}^i)\right)$$

where $r_j^i, 1 \leq j \leq n_i$, is a reference to another node, say N_j, and $\mathcal{R}_j^i, 1 \leq j \leq n_i$, is the region associated with N_j. Regions are not necessarily disjoint, that is, it is possible that some regions associated with nodes referred by the same parent node overlap.

In some tree-based access methods also internal nodes, in addition to leaf nodes, may contain references to searched data and the object that represents it. The corresponding (extended) node structure is the following:

$$N_i = \left(((p_1^i, O_1^i), r_1^i, \mathcal{R}_1^i), \ldots, ((p_{n_i}^i, O_{n_i}^i), r_{n_i}^i, \mathcal{R}_{n_i}^i)\right).$$

We will generally suppose that only leaf nodes contain pointers to real data, so we will use the simplified node structure. However, when needed we will mention the other possibility.

Let's suppose that N_p is an internal node and node N_c one of its child nodes. Let $\mathcal{R}_p$ be the region associated with N_p and $\mathcal{R}_c$ the one associated with N_c. As we have previously said, region $\mathcal{R}_c$ is included in region $\mathcal{R}_p$, that is if object $O \in \mathcal{R}_c$ then $O \in \mathcal{R}_p$.

The number of elements n_i in a node N_i, either leaf or internal node, and the type of regions associated with nodes depend on the specific access method.

The entry point to the tree is the *root node*. The root node is associated with a region representing the entire universe.

Notice that even if regions associated with different nodes might overlap, a reference to a record whose representation is contained in the intersection of two regions is typically stored just in one node.

When only leaf nodes contain references to real data, leaf nodes represent a partition of the whole data set. The strategies used to partition the data set and to bound elements of the partition with regions are specific for each access method.

Some tree-based access methods are designed to be maintained into main memory and do not take into consideration secondary memory storage. These are not suitable for large databases. Other access methods are primarily designed to be maintained

into secondary storage. In this case, in order to minimize disk access overhead, size of nodes of the tree corresponds to the size of a disk page. In this way reading a node of the tree corresponds to one disk access.

Most tree-based access methods are built using strategies that maintain them height-balanced. Therefore, the lengths of the paths from the root node to all leaf nodes are identical. In these cases, the length of the path from the root to the leaf nodes is called the *height* of the tree.

3.3.2 General similarity search algorithms

The hierarchical structure of tree-based access methods allows similarity search algorithms to ignore subsets of objects where qualifying objects cannot be found. Search algorithms traverse the tree structure top down, starting from the root, discarding paths corresponding to sub-trees where no qualifying objects can be found. Leaf nodes reached by search algorithms are exhaustively searched for qualifying objects.

For the sake of simplicity we suppose that data objects and references to searched data are only contained in the leaf nodes. However, it is straightforward to modify the algorithms in order to have a correct behaviour on tree structures where also internal nodes contain references to searched data.

The algorithms described in this section solve range and nearest neighbors queries defined in Section 2.4. The intuition behind these algorithms is the following. In case of range queries, only nodes associated with regions intersecting the query region $\mathcal{B}(O_q, r_q)$, where O_q is the query object and r_q the query radius, should be accessed. If the region associated with a node does not intersect the query region, all regions associated with nodes belonging to the sub-tree rooted at this node do not intersect

the query region and leaf nodes of the sub-tree do not contain objects that are covered by the query region. Therefore the entire sub-tree can be safely pruned. In case of nearest neighbors search, the behaviour is similar, since we will see that the nearest neighbors search algorithm is in fact defined as a range search algorithm where the query radius is reduced step by step. A query region with a large radius is used at the beginning. Then the radius is reduced step by step depending on the nearest objects currently found.

Similarity search algorithms on tree-based access methods traverse multiple branches of a tree to collect objects that satisfy the query. Accordingly, both algorithms use a dynamic queue of *Pending Requests*, **PR**, to temporarily store pointers to tree's nodes, whose bounding regions potentially contain qualifying objects, to be accessed in the next iterations of the algorithms. Each node corresponds to a different branch to be searched in the trees structure.

Range search algorithm

The response $\mathbf{range}(O_q, r_q)$ to the similarity range query of query object O_q and search radius r_q can be determined by the Algorithm 3.3.1.

The algorithm takes as input values the query region, composed of the query object O_q and the query radius r_q. It returns the set of pairs $\mathbf{range}(O_q, r_q) = \{(p_1, O_1), (p_2, O_2), ..., (p_l, O_l)\}$ where p_i is a pointer to a record, O_i is the object that represents it, and l is the number of elements retrieved.

The algorithm starts with an empty result set and a queue of pending requests **PR** containing only the pointer to the root of the tree (Step 1). Then the algorithm iterates while **PR** contains some nodes to be accessed (Steps 2 and 11). At each

Algorithm 3.3.1. **Range**
Input: query object O_q; query radius r_q.
Output: response set **range**(O_q, r_q).

1. Enter pointer to the tree root into **PR**; empty **range**(O_q, r_q).
2. While **PR** $\neq \emptyset$, do:
3. Extract entry N from **PR**. Suppose that N is bounded by region $\mathcal{R}$.
4. Read N.
5. If N is a leaf node then:
6. For each $(p_j, O_j) \in N$ do:
7. If $d(O_q, O_j) \leq r_q$ then $(p_j, O_j) \rightarrow$ **range**(O_q, r_q).
8. If N is an internal node:
9. For each child node N_c of N, bounded by region $\mathcal{R}_c$ do:
10. If $Overlap(\mathcal{B}(O_q, r_q), \mathcal{R}_c)$ then insert pointer to N_c into **PR**.
11. End

iteration, a node is extracted from **PR** (Step 3) and its content is read (Step 4). Notice that in case of secondary storage access methods, reading a node corresponds to one disk access. If the extracted node is a leaf node (Step 5) the algorithm checks if the contained objects are covered by the query region. Covered objects, along with the corresponding record pointers, are inserted in the result set (Steps 6 and 7). If the extracted node is an internal node (Step 8) it refers to other nodes of the tree, so the algorithm checks if these child nodes should be accessed. Each referred node is inserted in **PR**, to be examined in the next iterations, if the region, they are associated with, overlaps the current query region (Steps 9 and 10).

Algorithm 3.3.2. **Nearest neighbors**
Input: query object O_q; number of neighbors k.
Output: response set **nearest**(O_q, k).

1. Enter pointer to the tree root into **PR**; fill **nearest**(O_q, k) with k (random) objects; determine r_q as the max. distance in **nearest**(O_q, k) from O_q.
2. While **PR** $\neq \emptyset$, do:
3. Extract the first entry N from **PR**. Suppose that N is bounded by region $\mathcal{R}$.
4. If $Overlap(\mathcal{B}(O_q, r_q), \mathcal{R})$ then read N, exit otherwise.
5. If N is a leaf node then:
6. For each $(p_j, O_j) \in N$ do:
7. If $d(O_q, O_j) < r_q$ then update **nearest**(O_q, k) by inserting (p_j, O_j) and removing the most distant pair from O_q; set r_q as the max. distance of objects in **nearest**(O_q, k) from O_q.
8. If N is an internal node:
9. For each child node N_c of N bounded by region $\mathcal{R}_c$ do:
10. If $Overlap(\mathcal{B}(O_q, r_q), \mathcal{R}_c)$ then insert pointer to N_c into **PR**.
11. Sort entries in **PR** with increasing distance to O_q.
12. End

Nearest neighbors search algorithm

The response **nearest**(O_q, k) to the nearest neighbors query for query object O_q and k nearest neighbors, according to [HS95], can be executed by the Algorithm 3.3.2 in the optimum way, i.e. with the minimum number of accessed nodes.

There is a strong similarity between Algorithms 3.3.1 and 3.3.2. However we can highlight two main differences. First, the nearest neighbors algorithm maintains a dynamically shrinking query radius while the search radius (specified as input parameter) is constant for range queries. Second, the queue of pending requests **PR** does not assume any ordering for the range queries, while increasing distance to the query object is used for ordering in the approximate nearest neighbors algorithm – a

node close to the query object should be accessed first.

The algorithm for approximate nearest neighbors search takes as input values the query object O_q, the number of neighbors required k. It returns the set of pairs $\mathbf{nearest}(O_q, k) = \{(p_1, O_1), (p_2, O_2), ..., (p_k, O_k)\}$ where p_i is a pointer to a record, O_i is the object that represents it, and $d(O_q, O_i) \leq d(O_q, O_j), i < j$.

The algorithm starts with a result set initialized with k random objects from the data set, the current query radius r_q set to the maximum distance between the query object and the objects in the result set, and the queue of pending requests **PR** containing only the pointer to the root of the tree (Step 1). Then the algorithm iterates while **PR** contains some nodes to be accessed (Steps 2 and 12). At each iteration, a node is extracted from **PR** (Step 3). The content of the node is read only in case the region associated with it overlaps the query region (Step 4) elsewhere the algorithm exits since no other region in **PR** may overlap the query region, given the ordering used in **PR** (Step 11). If the extracted node is a leaf node (Step 5) the algorithm checks if the contained objects are nearer to the query objects than those in the current result set. In this case, these nearer objects are inserted in the result set, along with the corresponding record pointer, elements exceeding the k-th position of the updated result set are removed, and the current query radius r_q is set to the distance between the query object and the object that represent the k-th element in the result set (Steps 6 and 7). If the extracted node is an internal node (Step 8) the algorithm checks if pointed nodes should be considered or not. Each pointed node is inserted in **PR**, to be examined in the next iterations, if the region they are associated with overlaps the current query region (Steps 9 and 10). The entries in **PR** are sorted according to their distance to the current query object (Step

11); the distance of an object from a region is the nearest possible distance of any point inside the region from the object. Notice that given the dynamic behaviour of the query radius r_q, a node that is inserted in **PR** at Step 10 can be pruned at Step 4 since, after its insertion and before its extraction, the query radius might be reduced.

3.4 Specific tree-based access methods

In this section we describe some specific tree-based access methods. We have considered four different application areas and for each of them we have described the most popular tree-based access methods. The first application area is that of *one dimensional data*, where an absolute ordering among data can be conveniently exploited. This is a typical case in traditional database systems. B-Trees are the most popular tree-based access methods in this case. The second application area is that of *point or multi-dimensional data*, where data are represented by points in a multi-dimensional vector space. The point access methods described here are k-d-Trees and quad-Trees. The third application area is that of *spatial data*, where spatial objects like polygon, or curves, in addition to points, are managed. R-Trees are typically used as access methods in this case. Finally, the fourth application area is that of *metric or distance only data*, where no coordinate information can be exploited since only distances among data objects can be computed. M-Trees are a significant access method for this kind of data.

As previously stated, techniques developed in this thesis are defined for metric data. Specifically they were tested by using M-Trees as access method.

3.4.1 One dimensional access methods: B-Trees

B-Trees [BM72], and their extensions [Com79], especially B^+-Trees, are the most popular tree-based access methods. They can be used for one dimensional data, that is in those case where an absolute ordering among data can be exploited, and are widely used for high performance primary key accesses, in traditional databases. B-Trees are secondary storage structures, therefore, node's size corresponds to the size of a disk page. Each node, either leaf or internal node, may have several pointers to descendant (either nodes or data records).

The structure of a leaf node of a B-Tree is defined exactly as in Section 3.3.1. That is, it just contains a certain number of objects (keys) along with the corresponding references to real data (records).

Regions associated with nodes are disjoint adjacent intervals of the one dimensional space. The region $\mathcal{R}_j^i$ associated with the j-th child of the i-th node is

$$\mathcal{R}_j^i = [a_j^i, b_j^i)$$

Notice that, since intervals are adjacent,

$$b_j^i = a_{j+1}^i$$

so the structure of the node can be conveniently simplified by only storing the left bounds of the intervals.

Internal nodes of B-Trees, in addition to references to child nodes and associated regions, also contain references to data records, as in the extended internal node representation given in Section 3.3.1. Let us call p_j^i the j-th reference contained in node N_i. p_j^i points to the record identified by the left bound b_j^i of the j-th interval, that is, $O_i = b_j^i$ is the key of the pointed record.

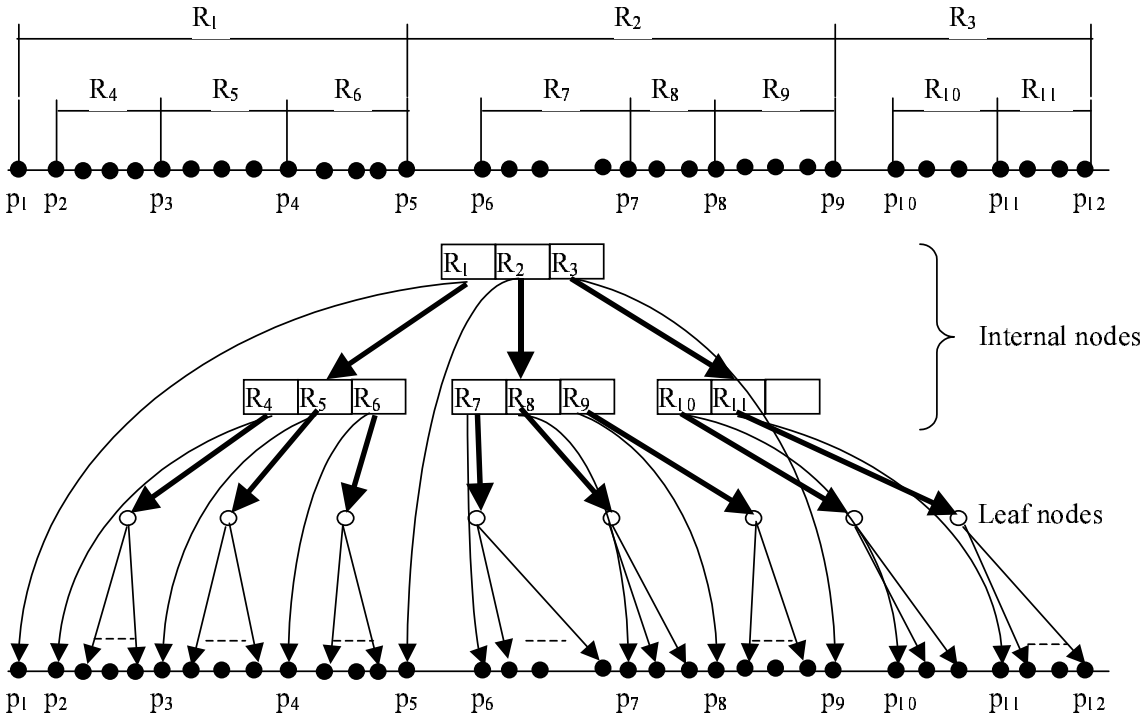

Figure 3.1: B-Trees structure example, supposing that the maximum number of entries in a node is three

Figure 3.1 shows the structure of a B-Tree and how intervals are organized.

B^+-Trees, simplify the B-Tree structure. In fact, in B^+-Trees only leaf nodes refer to data records, while internal nodes only refer to other nodes.

B-Trees and B^+-Trees are always height-balanced. The technique to guarantee this is to have the tree growing from the leaves to the root. Insertion first locates the leaf node where the new entry should be inserted. If the node is full, it is split in two. As a consequence of this, the entry referring the old node should be modified and a new entry, corresponding to the new node should be added in the parent node. If the parent node is also full, it is also split in two and splits may propagate up to the root of the tree. When also the root is split, a new level (a new root) is added to the tree, guaranteeing the height-balancing.

3.4.2 Point access methods: k-d-Trees and quad-Trees

Access methods described in this section are able to deal with points in a vector space, even if some of them were extended to deal also with spatial data as for instances polygons. We describe two different tree-based point access methods, namely k-d-Trees and quad-Trees, and discuss some of their extensions.

k-d-Trees

K-d-Trees [Ben75, Ben79] are binary search trees that exploit a recursive subdivision of the space. Let us suppose that our objects are vectors with dim dimensions. In every level of the tree a value (key) is used as discriminator for branching decision of a specific dimension of the corresponding vector space. The root node (level 0) discriminates for the first dimension. Nodes pointed by the root (level one) discriminate for the second dimension. And so on, up to the level $dim - 1$. If more than dim levels are needed, the process starts again from the first dimension. Space subdivision is obtained by $(dim - 1)$-dimensional hyperplanes. Each splitting hyperplane has to contain at least one data object. See Figure 3.2 for an example.

Leaf nodes just contain a reference to a record and the object (vector) that represents it, or they are empty in case that no objects are contained in the corresponding region. Of course empty leaf nodes can be materialized when needed, saving a lot of unused memory space.

The structure of internal nodes is slightly different than that described in Section 3.3.1:

$$N_i = \left((p_i, O_i), (r_l^i, \mathcal{R}_l^i), (r_r^i, \mathcal{R}_r^i)\right)$$

An internal node contains a single pointer to a record, along with its representation,

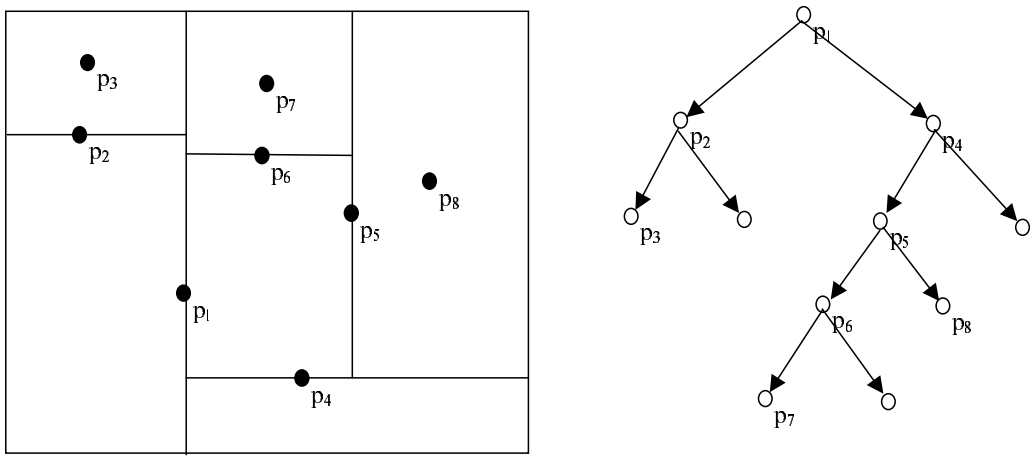

Figure 3.2: k-d-Trees structure example

and two entries. The left entry points to the left child node and contains a description of the region associate with it. The right entry, symmetrically points to the right child node and contains a description of its bounding region. Modifications of search algorithms for this modified structure are straightforward.

Let us suppose that N is a node at level lev of the tree, and object $O = (o_0, \ldots, o_{dim-1})$ is the object contained in it. Let $X = (x_0, \ldots, x_{dim-1})$ be the axis of our dim-dimensional vector space. The left region $\mathcal{R}_l$, associated with the left child node, is obtained by the intersection of the semi-space $x_{(lev \mod dim)} \leq o_{(lev \mod dim)}$ and the region $\mathcal{R}$ associated with the node N itself. Symmetrically, the right region $\mathcal{R}_r$, associated with the right child node, is obtained by intersecting the semi-space $x_{(lev \mod dim)} > o_{(lev \mod dim)}$ and the region $\mathcal{R}$. Notice that the hyperplane $x_{(lev \mod dim)} = o_{(lev \mod dim)}$ contains the object O.

It is straightforward to observe that the structure of a node can be conveniently simplified by only storing the value $o_{(lev \mod dim)}$, corresponding to the $(lev \mod dim)$ dimension, and the pointers r_l and r_r respectively to the left and right nodes.

The name k-d-Trees stands for k-dimension tree, where k is used as the dimensionality of the indexed space.

k-d-Trees have been designed as main memory access structures, but the extended k-d-Trees [CF80] has been modified for secondary storage environments. This scheme uses the k-d structure as a directory to data stored in pages, but only the leaves of the directory contain these page pointers.

A successful attempt to generalize the B-Trees and k-d-Tree ideas in one is presented in [Rob81] and called the k-d-B-Trees. In geometric terms, the data space is divided into a set of *dim*-dimensional boxes, which are also divided in boxes, etc., down to the depth of the tree. Instead of a set of one-dimensional points, each branch node in a k-d-B-Tree contains a set of regions. Each region is represented by the set of $2 \cdot dim$ coordinates which define its boundaries. The union of all regions in a branch node is the region which encloses them. The lowest level nodes, representing the "innermost" regions, contain pointers to sets of tuples pertinent to these regions. Alternatively, the leaf nodes may contain coordinates of their content points, plus another level of indirection to the actual tuples corresponding to these points.

quad-Trees

As previously discussed, k-d-Trees are binary trees where each node discriminate values of a single dimension of the vector space. The problem is that this data structure degrades its performance when the number of dimensions is high. An alternative possibility is to build a data structure where each node provides 2^{dim} descendants in order to ensure discrimination for all dimensions at the same time. This multi-way trees, extensively discussed in [Sam95], are called *quad-Trees* even if usually this term refers

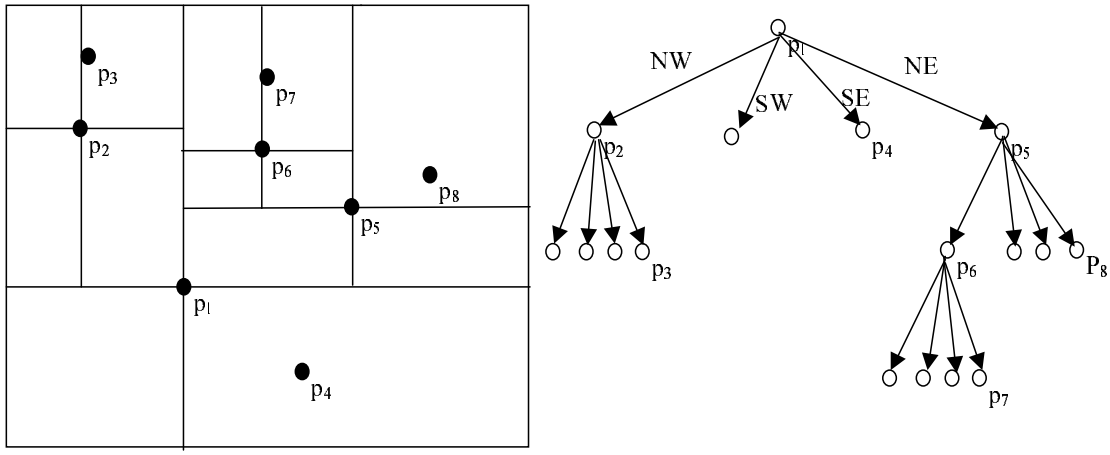

Figure 3.3: Point quad-Trees structure example

to the two-dimensional variant, where each node has four descendants. Quad-trees as k-d-Trees were also originally designed as main memory data structures.

In the two dimensional case, regions associated with nodes are rectangles. Let us suppose that N_i is an internal node. Its four descendants are associated with the four rectangles obtained by cutting the rectangle associated with N_i with an horizontal and a vertical straight lines. The four obtained rectangles are typically referred as the NW, NE, SW and SE (northwest, etc.) quadrants.

There are several variants of the quad-Tree structure. A very important one is the *point quad-Tree* [FB74]. In point quad-Trees, internal nodes have the following structure:

$$N_i = \left((p_i, O_i), (r_1^i, \mathcal{R}_1^i), \ldots, (r_{2^{dim}}^i, \mathcal{R}_{2^{dim}}^i)\right)$$

Similarly to k-d-Trees, each internal node contain a single reference to record along with the object that represents it. However, 2^{dim} entries, are included instead of just two. Each entry refers to a child node and contains a description of the region (hyper-rectangle) associated with the child node. See Figure 3.3 for an example.

As in k-d-trees leaf nodes either just contain one object, or they can be empty, in case no objects are contained in the corresponding region.

In a point quad-tree, objects (points in the *dim*-dimensional vector space) can be inserted consecutively. When a new object is inserted, first a point search is performed to check if the object already exist. If the object is not found, then it is inserted in the leaf node where the search stopped. If the leaf node already contains another object, then the associated region is divided into 2^{dim} subspaces, by using *dim* hyperplanes all containing the object contained in the leaf node. The former leaf node become an internal node containing 2^{dim} regions and referring 2^{dim} empty leaf nodes. The new object is inserted in the new leaf node referred by the entry containing the unique region that includes it.

Another popular variant of quad-Trees are the *region quad-Trees* [Sam95]. In this variant, regions are always partitioned regularly. That is, the 2^{dim} subspaces resulting from a partition are always of equal size. The hyperplanes used to cut regions do not necessarily contain the associated data object, as in point quad-Trees. For instance in a 2 dimensional vector space, they are such that they divide each side of the rectangular region in two segments of equals length. Some complex versions of region quad-Trees, as for instance the PM quad-Trees [SW85] or those described in [Sam88], are able to deal with spatial data in addition to points.

3.4.3 Spatial access methods: R-Trees

Spatial access methods are able to deal with spatial data such as polygons, curves, regions, etc. in addition to points. The most significant proposal of spatial access methods are the R-Trees, discussed in the following. Other spatial access methods

mainly refine the original design of R-Trees.

R-Trees were originally proposed in [Gut84]. They are height-balanced trees similar to B^+-Trees [BM72] (see also Section 3.4.1). Spatial objects are represented by their minimum bounding box. Therefore, objects are defined as follows:

$$O_i = (I_0, \ldots, I_{dim-1})$$

where I_i is an interval $[a, b]$ of the i-th dimension of the vector space, describing the extent of the spatial object along the i-th dimension. Of course, other representations for the bounding boxes are also possible.

Leaf nodes contain sets of objects (bounding boxes of spatial objects), along with references to real spatial objects.

Internal nodes are defined exactly as in Section 3.3.1. The region $\mathcal{R}_c$ associated with a child node N_c is the minimum bounding box including all objects contained in the leaf nodes of the sub-tree whose root is N_c. Regions associated with nodes at the same level of the tree may overlap, however each object is stored only in one leaf node even if it is covered by several regions. See Figure 3.4 for an example.

When a new object is inserted, the tree is traversed by choosing at each level the child node whose region needs the minimum enlargement to contain the object. If several children satisfy the same criterion, the one associated with the smallest region is chosen. Once arrived to a leaf node, if the corresponding region needs to be enlarged, it is adjusted appropriately, propagating the changes upward. If the leaf node is full, it is split and entries are distributed in the new and old node creating two new regions. These updates are also propagated upwards with eventually other splits in the internal nodes. Various strategies were proposed to minimize the overlap during insertion. Some were proposed in the original R-Trees paper [Gut84], other

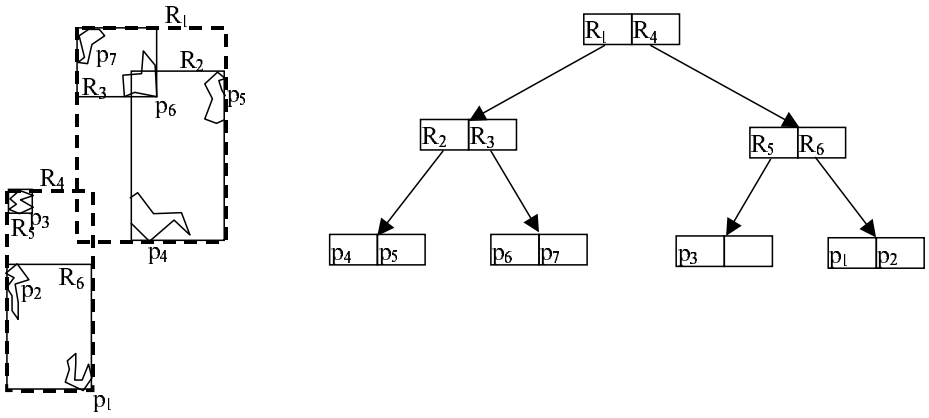

Figure 3.4: R-Trees structure example, supposing that the maximum number of entries in a node is two

were proposed later by other authors as for instance the *packed R-Tree* [RL85], or a packing techniques based on the use of the Hilbert Curve [KF93].

Zero overlap among regions is in general not obtainable. When a new data object (bounding box) that overlaps two regions is inserted, since we want to store it in just one leaf node, one of the two regions has to be enlarged, to include the object, and the consequence is that the enlarged region will overlap the other region. However, if we allow a data object, which overlaps more regions, to be stored in all overlapped leaf nodes, we can obtain non overlapping regions at the price of storing objects in several nodes. This is the main idea behind the R^+-tree [SRF87]. Another variation of the R-tree, called R^*-tree [BKSS90], does not change the properties of the basic structure, but rather has a more complex insertion algorithm that build the tree improving the overall performance.

3.4.4 Metric access methods: M-Trees

M-Trees [CPZ97] are secondary storage height balanced access methods. They organize data objects by creating partitions where set of objects are bounded by ball regions (see Section 2.3.2).

Data objects and references to real data records are stored in leaf nodes, whose structure is defined exactly as described in Section 3.3.1.

Internal nodes contain references to child nodes along with their bounding regions. In M-Trees bounding regions are ball regions defined as follows:

$$\mathcal{R}_i = \mathcal{B}(O_i, r_i)$$

where O_i is a reference objects, or routing object, used as center of the region and r_i is the radius of the ball region.

The ball region $\mathcal{B}(O_i, r_i)$ associated with the node N_i covers all regions associated with the nodes belonging to the sub-tree whose root is N_i and all objects contained in the leaf nodes of the sub-tree. That is, if $\mathcal{R} = \mathcal{B}(O, r)$ is a region associated with any node of the sub-tree, then $d(O_i, O) + r \leq r_i$. If O is any object belonging to a leaf nodes of the sub-tree, then $d(O_i, O) \leq r_i$. See Figure 3.5 for an example.

When a new object O should be inserted, the tree is traversed choosing in each level the child node that needs the minimum expansion (possibly no expansion) to contain the new object. In case of ties, the node associated with a region with the closest center is chosen. Finally the node is added to a leaf node. If it is full, it is split in two. Two new objects are chosen as centers of the two new regions, associated with the old and new node. Objects are distributed in the two nodes and their regions are defined by using the previously chosen centers and the distance from the centers to the farthest object in each node as corresponding radii. There are several strategies

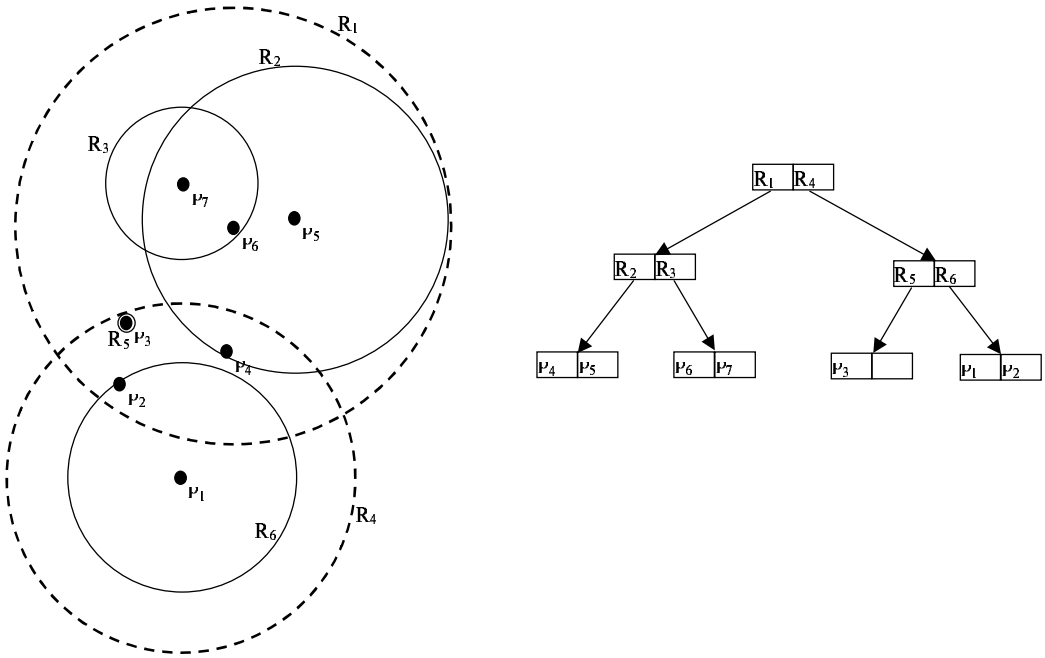

Figure 3.5: M-Trees structure example, supposing that the maximum number of entries in a node is two

that can be used to chose the two new representatives and to distribute objects in the two nodes. From various experiments, the authors say that the best strategy is trying a split that minimizes the maximum of the covering radii obtained. Notice that a split in a leaf node needs an insertion in the parent node. In case also the parent node is full then it has to be split too and the split can be propagated up to the root of the tree, as in B-Trees (see Section 3.4.1) and R-Trees (see Section 3.4.3).

As in R-Trees, regions of M-Trees may overlap and objects are stored in only one leaf node. A variation of the original design of the M-Trees that tries to minimize the overlap between regions are the *Slim-Trees* [TTSF00]. The basic structure of Slim-Trees is the same of the M-Trees. However, they use a new splitting algorithm based on Minimal Spanning Tree, a new algorithm to chose the most appropriate child node during insertion that leads to higher storage utilization, and a "Slim-down" algorithm

to be used in a post-processing step to reduce the sizes of the ball regions thus their overlaps.