Dynamic Indexing Using Hierarchy of Clusters, Updating Multiple Objects

Martin Gunia

December 22, 2005

University of Joensuu
Department of Computer Science
Master's Thesis

Abstract

Most metric indexes are designed to work with static data sets which cannot be changed once the index is built. This is impractical in situations in which the objects are often updated or their number changes frequently. Existing approaches are only capable of inserting or removing single object at a time. We present methods for both insertion and removal that operate multiple objects at the same time and show their advantage over single object manipulation experimentally. Moreover, we relax the accuracy of the methods with an error threshold parameter, which can be used for controlling the trade-off between speed of the update operations and quality of the index.

Table of Contents

1 Introduction **1**

1.1 Motivation for metric indexes 1

1.2 Applications 3

1.3 The theory of metric spaces 4

1.3.1 Metrics and metric spaces 4

1.3.2 Vector spaces 6

1.3.3 Example metric spaces 7

1.3.4 Query operations 7

1.3.5 Basic principles for metric indexing 8

1.3.6 Dimensionality and related issues 15

2 Related work **17**

2.1 Retrieval methods for vector spaces 17

2.2 Distance preserving methods 18

2.3 Distance based methods 19

2.4 Pivot selection 23

3 Dynamic indexing using unbalanced structures **24**

3.1 Dynamic capabilities of existing indexes 25
3.2 Dynamic vp-tree indexing 26
3.2.1 Insertion 28
3.2.2 Removal 32
3.3 Dynamic indexing using Hierarchy of Clusters 34
3.3.1 Multiple object insertion 36
3.3.2 Multiple object removal 37
3.3.3 Insertion with error control 38
3.4 The tree quality 40
3.5 Complexity analysis 40

4 Experimental results 44

4.1 Block size for updating multiple objects 45
4.2 Mean square error 47
4.3 Search comparison 50

5 Conclusions and future work 52

Bibliography 55

A Pseudocode 59

List of Figures

1.1 Boundaries of balls in 2 dimensional space $\mathbb{R}^2$ with covering radius r under L_1, L_2 and L_∞ metrics. 9

1.2 The model for indexing and querying on metric spaces. 10

1.3 a) Partition using 1 pivot, b) 3 pivots, c) Voronoi partition, d) Assymetric clusters. Objects marked c and p represent centers and pivots. 12

1.4 Comparing against more pivots to increase pruning. 14

3.1 Structure of internal DVPT node. 27

3.2 The effect of redistribution. Old boundaries are drawn in solid lines, the new boundary is dashed. Objects that are being moved are marked in grey. a) Shrinking class B by moving objects from B to C. b) Expanding class B by moving objects from C to B. 30

3.3 Splitting of node C. a) Splitting and inserting the new node in the tree. b) Updating the boundary. 2 levels of decomposition are shown for node C and C'. 31

4.1 Insertion cost in distance calculations for blocks of 10, 100 and 1000 objects. a) 10 dimensional clustered data, $h(n) = 1/4$. b) 15 dimensional uniformly distributed data, $h(n) = \sqrt{n}$. 45

4.2 Insertion cost in distance calculations for exact insertion with $\varepsilon = 0.01$ and blocks of 10, 100 and 1000 objects. a) 10 dimensional clustered data, $h(n) = 1/4$. b) 15 dimensional uniformly distributed data, $h(n) = \sqrt{n}$. 46

4.3 Removal cost in distance calculations for $\varepsilon = 0.01$ and blocks of 10, 100 and 1000 objects. a) 10 dimensional clustered data, $h(n) = 1/4$. b) 15 dimensional uniformly distributed data, $h(n) = \sqrt{n}$. 47

4.4 Error and construction cost for exact insertion for $\varepsilon = 0.01$, 0.05, 0.1. At the top 10 dimensional clustered vector space, at the bottom 15 dimensional space of uniformly distributed vectors. 48

4.5 Error and construction cost for object removal for different values of $\varepsilon = 0.01$, 0.05, 0.1. At the top 10 dimensional clustered vector space, at the bottom 15 dimensional space of uniformly distributed vectors. 49

4.6 Search efficiency for different values of $\varepsilon = 0.01$, 0.05, 0.1. and for non-exact insertion method. a) 10 dimensional clustered vector space, b) 15 dimensional space of uniformly distributed vectors. 50

A.1 The redistribution algorithm for HC-Tree. If the algorithm is called from within an `Insert_EXACT` function, it also calls `Insert_EXACT` instead of `Insert` and packs the node from which objects were removed to ensure the minimal error. 60

A.2 The trivial split algorithm. The algorithm rebuilds the whole splitted tree, possibly choosing new pivots. 61

A.3 The bulk insertion algorithm for Hierarchy of Clusters. Insert may return a new node in case of splitting, so we start the insertion by $root =$ `Insert` ($root$, O). 61

A.4 The exact insertion algorithm for Hierarchy of Clusters. The algorithm may return a new node in case of splitting, so we start the insertion by $root =$ `Insert_EXACT` ($root$, O, ε). 62

A.5 The merge algorithm. Merges left and right subtrees of A and replaces A with the result. 63

A.6 Remove algorithm. 63

A.7 Pack algorithm. Pack may return a new node if the root subtrees were merged. We start the algorithm by $root =$ `Pack` $(root, \varepsilon)$. 64

Chapter 1

Introduction

The aim of this work is to survey existing approaches to dynamic metric indexes and propose algorithms for insertion and removal on Hierarchy of Clusters. Additionally we consider updating multiple objects at once, which is more efficient than doing the same amount of work one by one. The thesis is organised as follows. The Chapter 1 provides basic information on metric spaces and the necessary theory. Chapter 2 presents existing methods for metric indexing. Chapter 3 is dedicated to dynamic indexes and propose new methods for inserting and removing multiple objects. Chapter 4 shows results from experiments conducted on the proposed algorithms. Conclusions and suggestions for further improvements are stated in Chapter 5.

1.1 Motivation for metric indexes

The most popular database systems these days are based on a relational data model. The model was first presented by E. F. Codd in 1970 [Codd 1970]. The main motivation behind his work was to offer more convenient and powerful models than the two opponents, hierarchical and network models. Founded on set and relational algebra, relational model is suitable for efficient implementations and has been successfully adopted in many commercial

solutions. Codd proposed to organise data in tables of fixed size *records* and assign each record a unique *key* as an identifier. Moreover, he sketched a simple set language that would be capable to retrieve subsets of the data using single operators. His work led to the design of the SQL language, which was adopted as a standard in 1986.

Relational databases are build around the concept of exact searching, where a record is included in the answer if its key matches the given query key (so called *querying by key*). Usually, one can also search for a range of keys. Optimal search algorithms exist, provided that there is a total linear ordering or at least an equivalence relation defined on the keys so that queries can be solved using a binary search method in $O(\log(n))$ worst case time, where n is the size of the database. Noticeably a weak point of this arrangement is that the database entries are accessed by an arbitrary identifier rather than the data itself. In a vast number of applications, however, searching on contents is essential and we speak about *querying by contents* or *querying by example*. The latter name comes from the fact, that we give an example object (query), that may not be in the database, and search by comparing the database against the example. The only way to search a relational database using the whole record is to declare the whole record a key. Unfortunately as the key becomes complex, both indexing and searching becomes too difficult and no longer to any benefit. Moreover, the requirements posed on a key strongly restrict the data types that can be used for indexing. A good example is a database of images, which cannot be meaningfully represented by a single key and for which no linear ordering is defined. Despite the fact that some workarounds to query by contents exist for specific domains (f.e. prefix search on textual databases), indexing complex data in general becomes very difficult or even impossible task for the relational model.

The querying on contents approach offers much higher level of freedom compared to the key based technique. Naturally, the set of relevant entries might not just contain the elements that match exactly but rather be defined by some relevance criterion. We may consider two situations. In the first one we have a complete query and we are looking for objects that are similar. In the second case, we only have a partial query and we aim to find the full record. Both share the same idea of retrieval of objects based on their proximity. Existing

solutions to proximity searching are based on the *metric space* model[1], where a distance function is defined over a set of objects. The objects can be of any type such as vectors, strings, images, documents or graphs which cannot be indexed by traditional databases. The distance function fulfils certain basic properties (see Section 1.3) that can be used for filtering out irrelevant objects.

Trivial way to evaluate a query would be to measure its distance to all objects but since the distance is usually expensive to compute, this is not practical for large databases. Therefore we look for a way to preprocess the database using the pairwise distances (i.e. build a *metric index*) so that at the query time we calculate as few distances as possible.

Metric indexes usually scale well with the size of the database and offer good performance to a certain extent. The most challenging obstacle in using metric indexes is their sensitivity to the dimensionality of the space. In general, the efficiency of searching in metric spaces drops drastically with increasing dimension and no algorithm can cope with data of insintric dimensionality higher than 20 without using some dimensionality reduction method.

Most metric databases are essentially static and the structure is fixed after the index was built. Update operations are very expensive on such indexes and very often the only way to add or remove an element is to rebuild the whole index from scratch. This may not be reasonable for many applications and some improvements towards dynamic operations have been done in [Ciaccia et al. 1997, Fu et al. 2000, Navarro and Reyes 2002; 2003].

1.2 Applications

We name a few domains, for which the concept of similarity searching is fundamental. A key to discover the range of application is to look at the data to be indexed and a set of queries to be answered.

Multimedia data types such as images, sounds or video cannot be meaningfully indexed using a traditional approach. Having a collection of images or songs, a typical query would

[1] Many of them are based on vector spaces as a special case of metric spaces.

be: "Find an image that is similar to this one.". Clearly, this cannot be done by assigning an image a key and then compare the keys. One workaround is to tag the images manually, i.e. to extract the features of the image and compare the sets of features. Metric indexes, on the other hand, allow indexing on the contents itself eliminating the error caused by the feature extraction. Another examples where metric spaces come to use is pattern recognition, video compression where metric indexes can be used for finding image blocks that differ from the previous frame, image compression where we try to find nearest match for parts of the image, audio retrieval or searching for similar DNA patterns.

1.3 The theory of metric spaces

In this section, we present basic theory that is necessary to understand how the distance information can be used for indexing and searching. At first, we introduce metric and metric space as the fundamental model for metric based indexing. We pay attention to a special class of metric spaces, so called vector spaces, for they are used used in a variety of applications such as spatial information systems or multimedia databases. We define basic operations on the metric spaces that are to our interest, namely range and nearest neighbour search. At the end of this chapter we speak about the importance of the dimensionality of the space and how it affects the efficiency of searching.

Let $\mathbb{X}$ denote the *universe* of all possible objects and its finite subset $\mathbb{U} \subset \mathbb{X}$ objects to be searched. $\mathbb{U}$ will be called the database or *dictionary*. $\mathbb{X}$ can consist of any objects, such as numbers, vectors, text strings but also graphs or images.

1.3.1 Metrics and metric spaces

The concept of proximity queries can be formalised using the metric space model. Metric space is a space of objects $\mathbb{X}$ where a distance between pairs of objects is defined. The mathematical abstraction of distance is called a metric and as such must follow some basic properties.

Definition 1.1. A function $d\colon \mathbb{X} \times \mathbb{X} \to \mathbb{R}$ is called a *metric* if it satisfies the following axioms:

A. $\forall x, y \in \mathbb{X}, d(x, y) \geq 0$ (non-negativeness)

B. $\forall x, y \in \mathbb{X}, d(x, y) = d(y, x)$ (symmetry)

C. $\forall x, y, z \in \mathbb{X}, d(x, y) + d(y, z) \geq d(x, z)$ (triangle inequality)

D. $\forall x, y \in \mathbb{X}, d(x, y) = 0$ if and only if $x = y$ (identity of indiscernibles)

A metric must be a nonnegative function. Most often, metrics are symmetric, but the symmetry criterion can be omitted in some cases which results in a quasi-metric. The *triangle inequality* reflects the fact, that one side of a triangle is always less than or equal to the sum of the lengths of the other sides or, in other words, that there is no shorter path than a straight one — from a geometrical point of view. Triangle inequality and symmetry are the two essential properties of metrics for any metric based indexing algorithm to work. Dropping the axiom D we obtain a pseudo-metric, allowing that two distinct objects exist at zero distance from each other. Depending on the domain of possible return values, the metrics are either *discrete* or *continuous*.

For clarity, we strictly refer to a metric when speaking about a distance function d in the rest of the work.

Definition 1.2. *Metric space* is a pair $(\mathbb{X}, d)$, where $\mathbb{X}$ is a set of objects and d is a metric.

A metric space is a set where a notion of distance between elements of the set is defined by the distance function d. Any subset $A \subset \mathbb{X}$ of metric space $(\mathbb{X}, d)$ is a metric space itself with the metric $d_A(x, y) = d(x, y)$ for all $x, y \in A$.

1.3.2 Vector spaces

If the objects are vectors, we speak about a *vector metric space* or *vector space* for short. Typically the objects are real numbered vectors but vectors of any types are possible[2]. Due to the geometrical nature of vector spaces, which is not present in general metric spaces, searching vector spaces takes advantage of the coordinate information. The algorithms are presented in the next chapter. Vector spaces usually come together with an L (Minkowski) metric.

$$L_k(\vec{x}, \vec{y}) = \sqrt[k]{\sum_i (x_i - y_i)^k} \tag{1.1}$$

The most widely used representative of the L family is the L_2 metric, known as the *Euclidean distance*. Euclidean metric corresponds to the natural notion of distance as the shortest straight path in a $\mathbb{R}^n$ space.

$$L_2(\vec{x}, \vec{y}) = \sqrt{\sum_i (x_i - y_i)^2} \tag{1.2}$$

Due to its higher computational complexity, several approximations exist. Generally they try to avoid the square root, which is expensive operation, using a polyhedral boundary to provide acceptable error/speed rate.

L_1 and L_∞ are other important L metrics. Besides being used in analysis, they are particularly useful because they are the lower respective higher bounds to the Euclidean metric and compute much faster. $L_\infty(x, y) \leq L_2(x, y) \leq L_1(x, y)$. The L_1 metric is also called *City Block* or *Manhattan distance* and corresponds to the sum of distances along the coordinates. L_∞ is the limit case of the equation (1.1) with k taken to the infinity, and can be computed as the maximum distance over all coordinates. L_∞ metric is sometimes referred to as the *Chessboard distance* as it is the distance the King can reach in one move.

$$L_1(\vec{x}, \vec{y}) = \sum_i |x_i - y_i| \tag{1.3}$$

[2]In linear algebra, any objects that are closed upon addition and scalar multiplication qualify for vector spaces

$$L_\infty(\vec{x}, \vec{y}) = \max_i |x_i - y_i| \tag{1.4}$$

For vector spaces, a whole group of searching algorithms exists, called *spatial access methods* and will be described in more detail in the next chapter.

These algorithms make use of the coordinate information present to partition objects in the space. These algorithms are, however very sensitive to the dimensionality of the space and the searching has exponential complexity on the dimension, making them impractical for dimensions higher than 10. In these cases, the general metric space can work better. In [Ciaccia et al. 1997] author compares M-tree with an vector space method R-tree and shows that M-tree outperforms R-tree in high dimensions. This phenomenon, known as the curse of dimensionality, is discussed further in Section 1.3.6.

1.3.3 Example metric spaces

The most common metric space is the space of real numbers $\mathbb{R}$ with the metric $d(x, y) = |x - y|$ and space of n-dimensional real number vectors $\mathbb{R}^n$ with an L metric.

Another example is a space of strings and *edit distance* function. Edit distance returns the minimal amount of characters that have to be inserted, deleted or replaced to transform one string to the other. It is useful in applications that need to determine how similar two strings are, such as spell checkers.

1.3.4 Query operations

Before we speak about query operations on metric spaces, we define few terms that are closely related and that will appear through the rest of the thesis. Let $x_0 \in \mathbb{X}$ and $r > 0$.

Definition 1.3. The set B:

$$B(x_0, r) = \{x \in \mathbb{X} \mid d(x, x_0) < r\} \tag{1.5}$$

is called an *open ball* (or ball) with center at x_0 and radius r.

Definition 1.4. The set $\overline{B}$:

$$\overline{B}(x_0, r) = \{x \in \mathbb{X} \mid d(x, x_0) \leq r\} \tag{1.6}$$

is called a *closed ball* with center at x_0 and radius r.

The parameter r is called the *covering radius* of B.

By querying we understand retrieving objects from the database, using an arbitrary object q, $q \in \mathbb{X}$ as a query object. Note that q does not have to be in the database. The result is a set of objects that are in a relation to q. The relation is determined by the distance function, see Figure 1.1. There are basically 3 interesting query operations, although one could argue about other candidates.

Definition 1.5. Given a query object q and a range r, *range query* (q, r) retrieves a set of objects that are within the distance r from the query object q. In other words, range query retrieves a ball centred at q with radius r.

Definition 1.6. Given a query object q, *Nearest Neighbour query* (q) retrieves an object u closest to q. That is $\{u \in \mathbb{U} \mid \forall v \in \mathbb{U} - u, d(u, q) < d(v, q)\}$.

Definition 1.7. Given a query object q and a threshold k, *k-Nearest Neighbour query* (q, k) retrieves a set of objects $S \subseteq \mathbb{U}, |S| = k$ that are closest to the query object q.
$S = \{u_i \in \mathbb{U} \mid \forall v \in \mathbb{U} - u_i, d(u_i, q) < d(v, q) \text{ for } i = 1 \ldots k\}$

1.3.5 Basic principles for metric indexing

Despite there are currently dozens of algorithms for searching in metric spaces, they have many things in common. Being based on the metric space model they rely merely on the distance information to drive the search. We do not consider special cases such as vector spaces in this section as they are not important topic in this thesis.

The common strategy of all algorithms is to partition the space into *equivalence classes* (further called *classes* for simplicity), which can be marked relevant or irrelevant as a whole

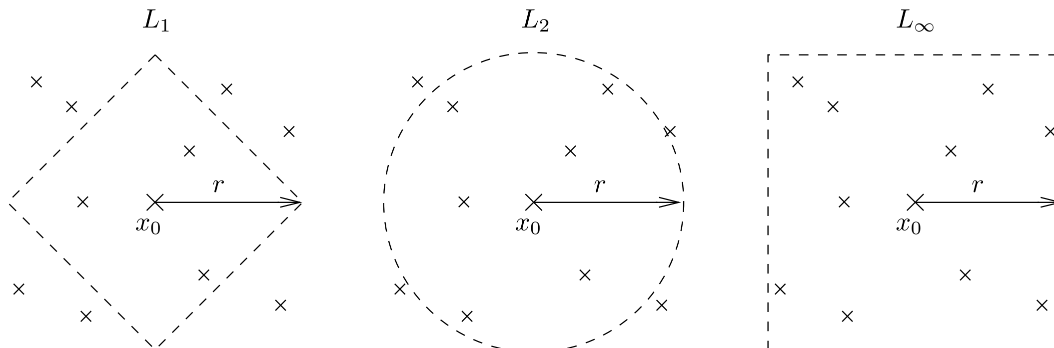

Figure 1.1: Boundaries of balls in 2 dimensional space $\mathbb{R}^2$ with covering radius r under L_1, L_2 and L_∞ metrics.

during searching. This way, the number of objects to be searched can be narrowed quickly. The equivalence classes are built around a specially elected object (or a small sets of objects) called *routing objects*. The searching process can be summarised into two steps:

1. Find the *candidate set* — a subset of the database, which may be relevant to the query.
2. Search the candidate sets exhaustively for the answer.

In the first step, the index is traversed and the query is compared against routing objects stored in the index, discarding all non relevant classes using an *exclusion criteria*. This includes distance calculations and some extra CPU time, together referred to as the *internal complexity*. The remaining classes are candidates for the answer and are searched exhaustively. This corresponds to the *external complexity* of the search. Routing objects play a special role in the first step and it is natural that their selection significantly affects the performance (see Section 2.4).

Most indexes are built recursively, suggesting a tree organisation. The decomposition continues until there is only one object to process in each subset. Additionally one may specify a constant b and stop the execution if there are less than b objects in every class. The objects are stored in *buckets* without any further processing. Controlling the granularity of the decomposition has it importance. Indexes with small buckets will have many internal

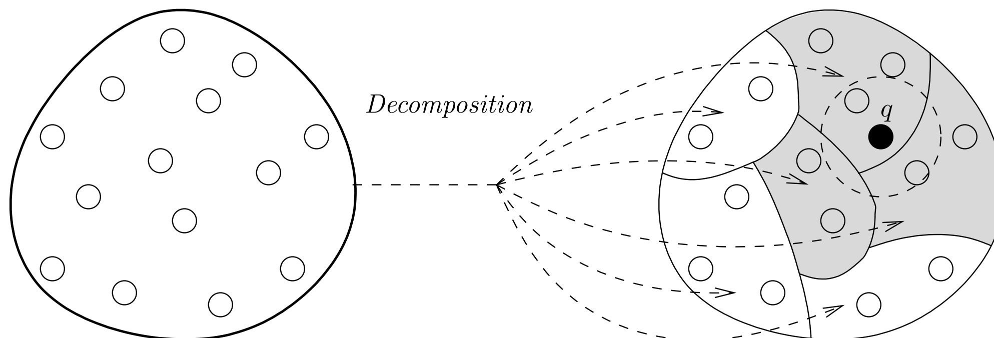

Figure 1.2: The model for indexing and querying on metric spaces.

nodes, hence higher internal complexity. The external complexity, on contrary, will be reduced.

A way to evaluate the quality of the decomposition and the exclusion criteria used is to calculate a *discriminative power* of the search algorithm with respect to the given query.[Chávez et al. 2001b] The discriminative power is tightly related to the external complexity and is defined as the number of the objects in the database relevant to the query divided by the number of objects that have to searched exhaustively. Discriminative power grows with decreasing external complexity and it is to the responsibility of the application programmer to find the optimal balance.

Partitioning strategies

Pivoting — given the set of objects $S \subset \mathbb{U}$ and a routing object (pivot) $p \in \mathbb{X}$, we consider distances $d(x, p)$ for all $x \in S$. Two elements x and y are considered equivalent (belonging to the same class) if $d(x, p) = d(y, p)$. We obtain a collection of disjoint spherical shells centred at p. Note that this is practical only if the distance is discrete or takes very few values. Otherwise the probability that two objects will be at exactly the same distance is minimal and we rather specify a lower and upper bounds on the distances allowed within each class. The same way, the elements x and y are considered equivalent if their distances to the pivot fall within the same range $r_{min} < |d(x, p) - d(y, p)| \leq r_{max}$. Clear disadvantage of using only one pivot is that the outer

rings tend to be very thin if the objects are distributed equally among the classes i.e. the index is balanced. The classes have very low *locality* as the distance of a pair of objects inside a class can be very high (see Figure 1.3a). The situation can be improved using k pivots [Bozkaya and Ozsoyoglu 1997] so that the above relations must hold for all pivots p_i, $i = 1 \dots k$, obtaining 2^k partitions. The locality increases significantly with increasing number of pivots as can be seen in Figure 1.3b.

Voronoi decomposition — given the set of objects $S \subset \mathbb{U}$ and k routing objects (centers) $\{c_i \in \mathbb{X}, i = 1 \dots k\}$ we assign each object to its nearest center[3]. That is we create one class for each c_i containing all objects $x \in S$ for which $d(x, c_i) < d(x, c_j)$ for all $j = 1 \dots k$, $j \neq i$. An example partition with four centers can be seen in Figure 1.3c.

Clustering — given the set of objects $S \subset \mathbb{U}$ and k routing objects $\{c_i \in \mathbb{X}, i = 1 \dots k\}$ we divide the objects into $k + 1$ classes, where each i-th class, $i = 1 \dots k$, contains all objects inside the ball $B_i(r_i, c_i)$. $(k + 1)$-th class contains the remaining objects. While the previous methods always create disjoint partitions, clustering does not. For an arbitrary object there might be one or more classes that qualify. It is interesting to remark that in case of overlapping classes the order in which they are traversed may matter. In such case we speak about *asymmetric* partition. If the traversal order does not matter (i.e. the object can be in any of the qualifying classes), the partition is *symmetric*. An example of an assymetric clustered partition is in Figure 1.3d.

Exclusion criteria

As mentioned before, at search time, the query is systematically compared against small subsets of the database and those that do not intersect with the query are discarded from further search. All the criteria follow from the triangle inequality as it is the only property that can be used for pruning in general metric spaces. Depending on the partitioning strategy

[3] A true Voronoi decomposition is only guaranteed to exist for Euclidean metric as the equidistant locus of two centers may not be a hyperplane in general case. We use the name for its graphical convenience.

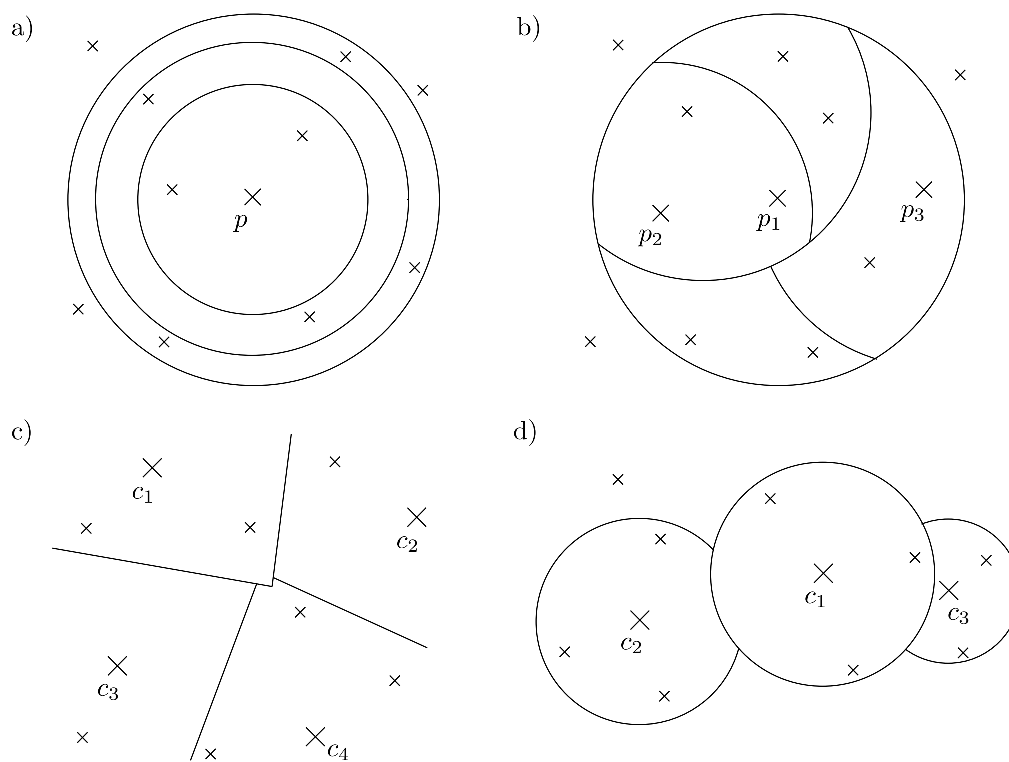

Figure 1.3: a) Partition using 1 pivot, b) 3 pivots, c) Voronoi partition, d) Assymetric clusters. Objects marked c and p represent centers and pivots.

used, the most common way to evaluate the intersection is to compare the query against the covering radii of the subsets or the hyperplanes between the neighbouring subsets.

Let S be a subset of U and p the routing object assigned to S.

Covering radius criterion bounds the set of objects S by considering a ball $B(p, cr)$ such that S is fully contained in B, hence $cr = \max_{u \in S} d(u, p)$. Given a query q and the search radius r, S can be discarded if

$$d(p, q) - r > cr \tag{1.7}$$

Covering ring criterion is a slight modification of the previous technique, which is in particular useful when p is far away from S. Covering ring defines both the upper and lower bounds on distances $cr_{min} = \min_{u \in S} d(u, p)$ and $cr_{max} = \max_{u \in S} d(u, p)$. S can be discarded if

$$d(p, q) + r < cr_{min} \text{ or } d(p, q) - r > cr_{max} \tag{1.8}$$

Hyperplane criterion considers the hyperplane between two neighbouring subsets of a Voronoi decomposition. If c is the center closest to q then the query of course intersects with the partition of c. The query does not intersect with a partition of c_i if it does not intersect with the hyperplane between c and c_i:

$$d(q, c) + r < d(q, c_i) - r \tag{1.9}$$

A way to decrease internal complexity without sacrificing the discriminative power is to use more routing objects for discarding irrelevant subsets.[Yianilos 1993, Brin 1995] Using the covering ring criterion, for example, we can discard all subsets for which (1.8) holds for any routing object considered. In general, a good idea is to use routing objects stored in the ancestral nodes [Yianilos 1993, Brin 1995] or on the same level [Brin 1995] of a tree as $d(q, p_i)$ has already been computed in the earlier stage of the search. The effect of using more pivots is illustrated in Figure 1.4. In the left part, S has to be processed for q_1. q_2, however, lies outside the ring and thus S can be skipped in that case. In the right part, S

does not have to be considered for both queries.

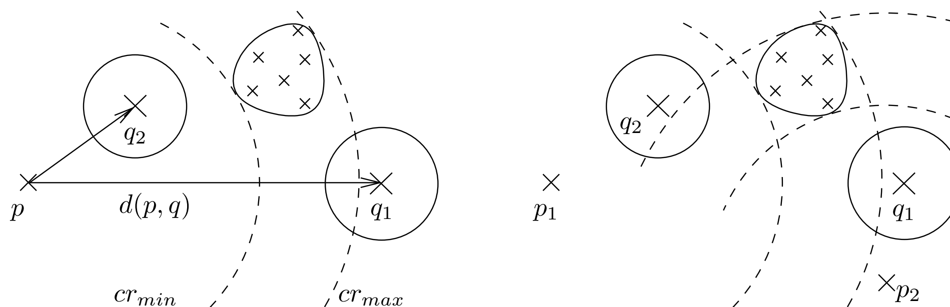

Figure 1.4: Comparing against more pivots to increase pruning.

Query complexity

In the first step, distance from a partition to a specially selected object is done and the partition is rejected using triangle inequality if it falls outside the query ball. As most of the indexes are hierarchical, the process is repeated until all partitions, that haven't been discarded yet, have been tested. The cost of this step is called the *internal complexity*. The partitions, that might contain relevant objects are searched exhaustively and the cost of the search is called *external complexity*. The total complexity to evaluate a query is then the sum of internal and external complexities. In other way the overall time to evaluate a query can be expressed as

$$T = N_D \times T_D + T_{CPU} + T_{I/O} \ , \tag{1.10}$$

where N_D is the number of distance calculations, T_D is the time per one distance calculation, T_{CPU} is the extra CPU time and $T_{I/O}$ is time needed for input/output operations.

The main task is to minimise the T. This is, however, dependent on the application domain and the optimal setup has to be proposed by the application developer. In many applications, a distance computation is very costly operation and thus the aim is to minimise the number of distance computations. The other factors can be neglected. In some cases, the index can suffer from very slow I/O access, mainly if the database is huge and does not fit in the main memory and have to be stored on the disk. Then we are trying to organise

the structure in order to minimise the number of page accesses. Most often the task of finding the optimal performance means finding the optimal trade-off between the number of distance calculations and the storage size (I/O cost).

1.3.6 Dimensionality and related issues

As mentioned before, the main obstacle for using metric indexes is their high sensitivity to the space dimensionality. Vector space indexing methods are exponentially dependent on the *representational dimensionality* of the data (that is on the dimensionality of the data representation, disregarding the "geometry" of the space) as the ball containing the answer grows exponentially with the dimension.[Chazelle 1994] This is true for all metric spaces and the phenomenon is generally known as *The Curse of Dimensionality.* The Curse of Dimensionality was first coined by Richard Bellman in [Bellman 1961], who relates the problem to the rapid growth of volume when increasing the dimensionality of a space. Bellman givens an example of 100 observations on 1 dimensional interval $\langle 0, 1 \rangle$. While in 1 dimension, 100 observations cover the interval quite well, considering a 10 dimensional space, 100 observations would be just few isolated points in a huge empty space. To get the same coverage, one would have to use 10^{20} observations instead. Another way to look at the dimensionality problem is to consider the histogram of distances between objects in $\mathbb{U}$. [Chávez et al. 2001b, Brin 1995] refers to the histogram as a fundamental analytical measure of the dimensionality related properties in generic metric spaces, independent of their nature. High dimensional spaces have a histogram concentrated around the mean, with low variance, suggesting that the distances between objects are very similar and therefore carry less information.

The representational dimensionality is overcome in general metric spaces as the distance information uncovers the geometry of the space. The true dimensionality is also called the *intrinsic dimensionality.* In practise, insintric dimensionality can be much lower as the data does not occupy the whole space but rather agglomerates in clusters. [Chávez et al. 2001b] gives an example of a hyperplane embedded in a 50 dimensional space whose representational dimensionality is 50 but insintric dimensionality is 2. Formally we define

the insintric dimensionality of a metric space ρ as

$$\rho = \frac{\mu^2}{2\sigma^2} \tag{1.11}$$

where μ is the mean and σ^2 the variance of the histogram of distances of the space.

Algorithms that use Voronoi-like decomposition, try to overcome the high insintric dimension by increasing the number of centers used for the decomposition. This approach, however, requires a lot of memory or disk space and poses a limit on the amount of centers that can be used. Additionally, if the data is organised on disk, large amount of centers will increase the disk page accesses, slowing down both search and update operations.

In [Chávez and Navarro] authors show the disadvantage of using partitions with equally sized classes for spaces with high insintric dimensionality. The problem is related to the exponential growth of volume as the dimension increases. To have equal volumes, the classes farther from the center have to be very narrow compared to classes lying closer. While the effect is negligible in low dimensional spaces, in high dimension the query ring may intersect with many classes or, in the worst case, with all of them, decreasing the efficiency of searching. Some solution to this problem are List of Clusters and Hierarchy of Clusters described later, which use unbalanced partitions.

Another solution to indexing high dimensional spaces are the probabilistic algorithms. The probabilistic framework presented in [Chávez and Navarro 2001, Chávez and Navarro 2003] can be applied to any indexing algorithm and significantly improves the search performance at the cost of very few errors. The idea is to relax the border of the result by a confidence ε, which defines how much can the outcome of the query differ from the correct answer. Searching for a query q with radius r, the algorithm discards all objects u for which $|d(u,p) - d(p,q)| > r/\beta$, where p is a pivot and β is a stretching factor. The main goal is to find a maximum β so that the probability of missing an answer is at most ε.

Chapter 2

Related work

In this chapter we present existing metric indexes. This survey is based on the comprehensive work of [Chávez et al. 2001b]. We divide the survey into three parts. First we introduce retrieval algorithms for vector spaces for their importance in practice, then we briefly present algorithms that retrieve objects by mapping the original space into a vector space. At last we focus on the metric indexes, which are central in this work.

2.1 Retrieval methods for vector spaces

Multidimensional access methods (also called spatial access methods — SAM) provide efficient searching in spatial databases. Spatial database is a collection of points or, more generally, shapes embedded in a $\mathbb{R}^n$ space[1]. Shapes can be approximated by a single point and a covering radius or by a polygonal boundary. In case of points, the space is indeed a vector space, usually grouped with an Euclidean distance. A typical operation on a spatial database is searching for all objects in a given area, either fully or partially contained. Efficient spatial indexes are essential in geographical and CAD applications as well as multimedia databases, which often represent indexed objects as points in multidimensional space

[1] $\mathbb{R}^n$ is the most common but in general, vectors of any data type are possible

for efficient retrieval.

A general solution to multidimensional indexing is a grid file presented in [Nievergelt et al. 1984]. Given a set of n dimensional vectors, grid file defines a search space of n dimensions where each dimension represents an ordering on the corresponding component of the data. This is, in fact, identical to indexing over all fields in a relational database and does not overcome the restrictions such as the data types that can be indexed. For this reason, grid file is considered a type of disk data organisation, rather than an index itself. Amongst the other indexes, the most popular are kd-trees [Bentley 1975; 1979], R-trees [Guttman 1984] (and its modifications: R^+-trees, R^*-trees) and X-trees [Berchtold et al. 1996]. All recursively partition the space into disjoint subspaces in order to discard regions that do not contain the query. K-d tree uses hyperplane partitioning along the coordinates, very simple yet effective technique. R-tree divides the space into hyper-rectangles.

Unfortunately, the search complexity of the above algorithms is exponentially dependent on the dimensionality of the space. High dimensional data with low insintric dimensionality is common to many real applications, programmers sometimes resort to general metric spaces where the insintric dimensionality shows up without any need for an additional workaround.

2.2 Distance preserving methods

Distance preserving method try to transform the general metric space into a vector space so that the objects can be retrieved using a SAM method. The technique of mapping from the general metric space to a vector space is called *multidimensional scaling* (MDS). MDS reduces each object in the original space into a representative vector so that the distances in the target space are preserved as much as possible. The two reasons for using MDS. The first is efficiency — complex analysis on high dimensional spaces is inefficient (or even impossible), reducing the dimensionality while preserving distances allows us to perform operations faster. The second reason is visualisation of highly dimensional data. Reducing to 2 or 3 dimensional vectors can help to better understand the structure of the analysed space.

In [Kruskal and Wish 1978] author defines a *stress* function that ranks the dissimilarity between the pairwise distances using a stress function. The algorithm first guesses the target vectors and then tries to minimise the stress. The algorithm uses $O(n^2)$ distance calculations.

Probably the most known example is the FastMap algorithm [Faloutsos and Lin 1995] which offers similar quality of the mapping as the previous algorithm, but only takes $O(nk)$ distance calculations.

In [Fu et al. 2000] authors show experimentally that the accuracy grows with the dimensionality of the target space. In other words, to have accurate mapping, we need the target space to be of high dimensionality also, speaking clearly in favour of distance based methods presented in the next section.

2.3 Distance based methods

Nowadays, it is common for multimedia databases to index their contents, f.e. images, using feature vectors. However, it is not easy and often not possible at all to transform the original data into a vector of limited size without a loss of accuracy in searching. Moreover, some data types, such as text, are completely resistant to feature extraction and cannot be transformed by any means. Metric indexes overcome this problems by using the distance information only rather than relying on other way of representation.

Distance based indexes are the central topic in this work. We introduce the indexes in chronological order, as they appeared, because many of them are in fact closely related or just slight modifications of previous algorithms. We try to point out the similarities. Attention is payed to the VP-Tree, List of Clusters and Hierarchy of Clusters, because the indexes will be referenced in the next chapters.

The first general solution for searching metric spaces was given in [Burkhard and Keller 1973]. They propose a tree, hence called *Burkhard-Keller Tree* (BKT), that partitions the space using one pivot per node. BKT assumes discrete distance function as each partition consists of objects whose distance to the pivot is equal, i.e. for each distinct distance there

will be one partition. The process is repeated recursively. A slight improvement of BKT is the *Fixed Query Tree* [Baeza-Yates et al. 1994] which uses one pivot per level of the tree. Authors claim that the tree can save some distance computations during backtracking and show this fact experimentally. In [Chávez et al. 2001a], *Fixed Query Array* is presented, which is no more than a compact representation of FQT, which corresponds to the elements of the FQT stored in an array when traversing the tree in-order. Despite BKT and FQT, FQA quantizes the distances to fit in a predefined number of bits, making the index as compact as possible. This allows to use more pivots in sacrifice of the precision, which may give better performance than using the original distance.

In [Uhlmann 1991] Uhlmann presented a structure called Metric Tree. Later, the same idea was extended in [Yianilos 1993] and called the *Vantage Point Tree* (VPT). VPT is a binary tree built recursively by taking an element as a vantage point and calculating the median of distances from all elements to it. Elements whose distance is smaller or equal to the median are put in the left subtree and elements whose distance is bigger are put into the right subtree. The construction continues recursively for both subtrees. Yianilos extends the tree by storing the upper and lower bounds in each internal node for its corresponding subspace as viewed from the ancestor vantage points. Since the distances have already been computed, the construction cost remains $O(n \log(n))$. Searching the tree is similar to BKT. The distance from the query to the vantage point is measured and the search enters all subtrees that intersect with the query ball until a leaf node is found. Some subtrees can be discarded without having to measure the distance by evaluating the bounds stored in the node for intersection with the query. The tree can be generalised to m-ary tree (MVPT) by taking $m - 1$ uniform percentiles instead of the median or using more vantage points per node, as suggested in [Bozkaya and Ozsoyoglu 1997]. The first technique slightly improves the search efficiency compared to the binary tree but not in all circumstances. In high dimensional spaces, for example, the effect is negligible as the outer rings tend to be very thin and the search will likely enter many subtrees. Authors show that using more pivots improves the search efficiency by 20% – 80%.

Uhlmann also suggests taking two centers and dividing the space using a hyperplane between the centers, hence the name *Generalised Hyperplane Tree* (GHT). The advantage,

compared to VPT, is that the decomposition is symmetric in volume, so that the volumes of so created partitions are equal. The search algorithm uses the hyperplane as the pruning criterion. Brin proposed to use m centers to obtain better partitioning of the space in [Brin 1995]. Result is the *Geometric Near Neighbour Access Tree* (GNAT). In addition to the hyperplane criterion, GNAT also uses the lower and upper boundaries of each partition with respect to the other centers to save distance computations during searching. This idea is very similar to that of Yianilos.

An algorithm which is different from the others is the *Approximating Eliminating Search Algorithm* (AESA) by [Ruiz 1986]. AESA does not rely on hierarchal decomposition but uses a matrix of precomputed pairwise distances between all objects in the database. At search time, the algorithm picks an object $p \in \mathbb{U}$ from the database and discards all objects $u \in \mathbb{U}$ that do not satisfy $d(p, q) - r \leq d(u, p) \leq d(p, q) + r$. The process is continued until no objects can be eliminated and the rest is searched exhaustively. Because all $d(u, p)$ distances are precomputed, only one distance calculation is needed at each step. AESA performs remarkably better than any other index in the terms of distance computations, but its disadvantage is the $O(n^2)$ space consumption, which makes it impractical for larger databases. [Micó et al. 1994] overcomes the problem by taking k sample objects and storing the distances to all remaining objects so that the index occupies $O(kn)$ space. This simplification comes from the assumption that k steps are enough in most situations to achieve efficient pruning.

Another algorithm that does not divide the space using centers is the *Spatial Approximation Tree* (SAT) [Navarro 2002]. The tree is constructed as follows. An element p is selected at the root of the tree and is connected to a set of neighbours $N \subset \mathbb{U}$ such that every object in N is closer to p than to any other object in N. The remaining objects (not in N) are assigned to their closes object from N and processed recursively. Searching is also different from the other indexes. When searching with zero radius, we simply move to the neighbour, which is closest to the query. If the radius r is greater than zero, we pretend we are searching for an unknown element q' whose distance from q is at most r. Therefore the search may enter more neighbours, not only the closest one. If c is the closest neighbour to q, we enter into all neighbours $u \in N$ for which $d(u, q) - r \leq d(c, q) + r$ satisfies.

M-Tree presented in [Ciaccia et al. 1997] divides the objects by their distance to the centers chosen at each node. Each class within the partition stores its center and covering radius. The search algorithm enters into all subtrees that qualify for the answer. Due to the symmetric partitioning scheme, and because all objects are stord in leaves, M-tree handles both insertion and removal operations very efficiently.

In [Chavez and Navarro 2000, Chávez and Navarro] authors propose List of Clusters (LC), a linear clustering technique that groups objects in the space into balls of defined radius. The construction starts by choosing a center $c \in \mathbb{U}$ and a radius r_c and divides the set $\mathbb{U}$ into two subsets

$$I_{\mathbb{U},c} = \{u \in \mathbb{U} - \{c\}, d(u,c) \leq r_c\}$$

$$E_{\mathbb{U},c} = \{u \in \mathbb{U} - \{c\}, d(u,c) > r_c\},$$

where I is the set of internal objects lying inside the ball $B(c, r_c)$ and E stands for the set of external objects, lying outside the ball $B(c, r_c)$. The set I is stored in a bucket while the set E is processed recursively until E contains no objects. The result is a list of triplets (c_i, r_i, I_i). It is important to note that the partition is asymmetric and the first center chosen has a preference in case of overlapping balls. The search algorithm starts at the beginning of the list ($i = 1$) and measures the distance from the query q to the i-th center $d(q, c_i)$. c is added to the result if it lies inside the query ball (q, r), that is if $d(q, c_i) \leq r$. If the query ball has an intersection with the ball $B(c_i, r_i)$, the bucket I_i is searched exhaustively. The search continues for $i + 1$ until the end of the list is reached. The search does not always have to scan the whole list. Due to the asymmetry, the process can be stopped if the query ball is fully contained inside $B(c_i, r_i)$.

We can see List of Clusters as a special case of tree, similar to the VP-Tree. The main difference is that LC is highly unbalanced, because I is typically much smaller than E. Authors show both analytically and experimentally that unbalanced structures work better in high dimensions.

List of Clusters can be easily turned into *Hierarchy of Clusters*[Fredriksson 2005] by processing both sets I and E recursively, until each contains only one object, which is stored

in a leaf, or b objects stored in a bucket. Rather than relying on a fixed size or fixed radius, function $h(n)$ is used used to control the size of the set I, hence

$$I_{\mathbb{U},c} = \{u \in \mathbb{U} - \{c\}, d(u,c) \leq h(|\mathbb{U}|)\}$$

$$E_{\mathbb{U},c} = \{u \in \mathbb{U} - \{c\}, d(u,c) > h(|\mathbb{U}|)\}$$

Depending on the $h(n)$ the tree can be either balanced $h(n) = n/2$ or unbalanced. In a way HC can be seen as a generalisation of the simple VP-tree, which always creates balanced partition. Searching the Hierarchy of Clusters is identical to searching the VP-Tree.

2.4 Pivot selection

Most algorithms, that use routing objects for indexing select the objects randomly. However, it is known that the pivot selection significantly affects the search time and some heuristics better than the random selection have been presented independently in [Brin 1995, Chavez and Navarro 2000, Chávez et al. 2001b]. In general they try to use objects that are far from each other. In [Brin 1995], for example, authors suggest to choose objects farthest from each other using a greedy algorithm so that they are likely to be in the centers of the partitions. Centers bunched up nearby each other would not be of much use since their distances to a random object would be similar. [Chavez and Navarro 2000] suggests, for each new center, picking an object that maximises the sum of distances to previous centers so as to minimise the overlap of clusters. More complete work on pivot selection is presented in [Bustos et al. 2003]. They define an efficiency criterion to estimate the capability of a set of pivots to filter out objects outside the query. The criterion is based merely on the distance distribution of the data set. They show that good pivots tend to be outliers, hence objects that are far from each other and far from the rest of the objects in the database but not all outliers are good candidates for pivots.

Chapter 3

Dynamic indexing using unbalanced structures

Most metric indexes are designed to work with static data sets which cannot be changed once the index is built. This is impractical in situations in which the objects are often updated or their number changes frequently. Trivial solution to the problem would be to generate a new index every time the data is changed but this is not efficient, considering that such databases typically consist of tens of thousands of objects. Modifying even a small portion of the data set would require rebuilding the whole index from scratch. For this reason it is useful to look for indexes capable of efficient insertion and removal of objects. Note that an object in the data set can be modified by removing from the database and inserting it's updated version. The basic requirements on a dynamic index can be summarised as follows:

1. The modification is local — in order to ensure efficiency, the modification should not require global reorganisation of the index. The cost should be proportional to the amount of objects inserted or removed.

2. The quality of the decomposition should not decrease after subsequent modifications or stay within a defined bound.

Inserting and removing is a difficult task for hierarchal indexes, which often have to reorganise the partitions to maintain their quality. The difficulty then depends mainly on the partitioning scheme used. Partitions that create disjoint classes of equals sizes, which are the most commonly used in metric indexes (for example BKT, VPT or Hierarchy of Clusters), are very resistant to any updates. This is because the partitioning in lower levels of the hierarchy depends on the partitioning in the upper levels, closer to the root of the hierarchy tree, and changes to any node have to be propagated downwards. Non-hierarchal indexes usually handle both operations without difficulties.

3.1 Dynamic capabilities of existing indexes

Among the indexes presented in Section 2.3 the only representatives that can truly handle both insertion and deletion are FQT (and it's modifications FQHT and FQA), AESA (and LAESA), GHT, GNAT and M-Tree. FQT has no internal nodes and the decomposition is predetermined by the set of pivots, therefore an object can be added or removed from the tree without any structural reorganisation. However, after a large amount of objects being inserted or removed, pivots may not fit to the data anymore, leading to increase in performance. In such situation, a global reorganisation may be needed. AESA and LAESA can handle updates without any difficulty since they are just matrices of distances between pairs of objects. GHT and GNAT do not rely on global statistics (such as median for VPT) so that any reorganisation happens in leaf nodes only. Like for FQA, the centers may become inappropriate after a while and global restucturalization may be needed. Removing a center is also a problem.

M-Tree presented in [Ciaccia et al. 1997] takes advantage of symmetric partition which allows fast insertion and removal of objects. The inserted object is placed in the subtree for which the increase in radius is minimal or zero in ideal case. When the leaf node is reached, the object is stored and if the node overflows, it is split into two and one node is promoted upwards. Similar strategy is used for removing objects, where an underflown node is merged with its neighbour and the merging is propagated upwards until needed.

SAT does not support any insertions or deletions as the whole dataset has to be known in advance to find the candidate neighbours. Some work on dynamic SAT (DSAT) has been done in [Navarro and Reyes 2002]. Deletions were furthermore improved in [Navarro and Reyes 2003].

VP-Tree and MVP-Tree suffers from relying on median (or uniform percentiles for MVPT) of distances from the vantage point so that inserting or removing objects from leaf nodes affect the decomposition at higher levels (closer to the root). In [Fu et al. 2000] authors propose a modification of VPT called Dynamic VPT (DVPT), which supports insertion and deletion. The technique resembles that of M-Tree but requires more careful reorganisation since it uses disjoint partitions and may lead to global reorganisation as well.

List of Clusters handles insertion and removal easily for clusters of fixed radius because the partitions are not modified. The optimality of the chosen radius may be affected after many objects have been inserted or removed and therefore a periodical rebuilding of the structure is suggested. If the size of cluster is fixed, the update operations are more complex and resemble that of DVPT.

3.2 Dynamic vp-tree indexing

In this section we present a dynamic index suggested in [Fu et al. 2000] that derives from the VP-Tree, hence called DVP-Tree. The index supports both insertion and removal of a single object with reasonable efficiency. In the worst case, however, insertion of an object may result in rebuilding the whole tree, the matter is discussed later. Taking advantage of the similarity of the two indexes, we use their work as a basis for our algorithms. In fact, the algorithms for inserting and removing a single object from HC are just slight adaptations of the DVP-Tree algorithms that treat the balance rather as a parameter than a fixed state.

The structure of DVP-Tree is nearly identical to the VP-Tree. Consider a finite set of objects S and a parameter $M, M \geq 2$. In each internal node, the tree stores one routing object $vp \in S$ (vantage point) and divides the data set into M classes $S_i, i = 1 \dots M$ of roughly equal sizes depending on their distance to the vantage point. The boundary distance

P	pointer to the parent node
vp	vantage point
$\mu_i,\ i = 1 \ldots M{-}1$	i-th class boundary
$child_i,\ i = 1 \ldots M$	i-th child pointer

Figure 3.1: Structure of internal DVPT node.

of i-th class is denoted by μ_i so that for each $s \in S$, $\mu_{i-1} < d(s, vp) \leq \mu_i$.[1] The structure of an internal node is shown in Figure 3.1. Note that since the classes are constructed on the whole set S, the vantage point is also contained in one of the sets S_i. Therefore the tree stores all objects in the leaves, which results in somewhat taller trees, but has it's reason. Having all objects in leaves, one can easily remove an arbitrary object from the tree without destroying it's functionality. Otherwise it's difficult to remove a routing object because there is a part of the tree depending of it. One solution would be to give each routing object a tag marking whether it has been deleted or not. Upon deletion, the tag would be set to *true* and the object would remain in it's place. The negative effect of this approach is that quality of the tree decreases after many routing objects have been deleted. Duplicated routing objects, present the same efficiency drawback but do not require any special treating while deleting objects. In practise, the decrease of efficiency caused by routing objects being replicated in leaves is only visible in low dimensional spaces. The structure of DVP-Tree node is shown in Figure 3.1.

Let us also define several operations on the nodes of the tree, which will be used throughout the thesis:

Definition 3.1. If A is a node, then $[A]$ denotes the set of objects held in the tree rooted at A. For leaf node $[A]$ contains all objects stored in the bucket of A. If A is an internal node with m children then

$$[A] = \bigcup_{i=1}^{m} ([A.child_i]) \tag{3.1}$$

Definition 3.2. If A is a node, then $count(A)$ denotes the the number of objects held in the tree rooted at A. For leaf node $count(A)$ equals to the number of objects in the bucket

[1] Provided that $\mu_0 = 0$ and $\mu_M = \infty$

of A. If A is an internal node with m children then

$$count(A) = \sum_{i=1}^{m} (count(A.child_i)) \tag{3.2}$$

Definition 3.3. Let $level(A)$ be the level of the node A, i.e. the distance of the node to the leaf level. Level of a leaf node equals to 0.

Definition 3.4. If A is a node, then $capacity(A)$ denotes the the maximum number of objects that can be held in the tree rooted at A. Let *bucketsize* be the capacity of a leaf node and M the maximum number of children an internal node can have. Then

$$capacity(A) = bucketsize \cdot M^{level(A)} \tag{3.3}$$

Definition 3.5. If A is a node, then $free(A)$ denotes the size of unoccupied space under A. $Free(A)$ is defined for convenience only and can be calculated as

$$free(A) = capacity(A) - count(A) \tag{3.4}$$

3.2.1 Insertion

Being a hierarchal index that partitions the space in a top-down manner using classes of fixed size, DVP-Tree relies on complex reorganisation techniques in order to maintain the quality of the partition and a global reorganisation may result. To avoid global reorganisation until necessary, the update algorithms look for the smallest portion of the tree that needs structural changes. To insert an object, the reorganisation is forced when there is no place to store the inserted object, using two methods: *redistribute* and *split*.

Redistribution does not create any new nodes in the tree but moves objects between adjacent siblings in order to make the space for insertion available. Assume that the new object is being inserted into k-th child of a node A. Let's also assume that the k-th child cannot accommodate any new object ($capacity(A.child_k) = count(A.child_k)$) but there exist at least one child l which can ($capacity(A.child_l) > count(A.child_l)$). We move objects

from k-th child to it's nearest non-full sibling l so that the new object can later be inserted. For simplicity, we will only focus on the situation when the redistribution happens between two adjacent trees, i.e. $l = k + 1$ or $l = k - 1$, even though in general, we might need to move objects from a node to any of it's siblings. This can be achieved by subsequent redistributions. The algorithm for redistributing between adjacent k-th and (k+1)-th nodes follows.

1. Calculate the average amount of objects under both nodes.

$$\overline{n} = \frac{count(A.child_k) + count(A.child_{k+1})}{2} \tag{3.5}$$

2. If $count(A.child_k) > \overline{n}$ we move objects outside k-th child (*shrinking* k-th class). Collect objects from k-th child plus the inserted object and divide the objects into two subsets SS_1 and SS_2 so that SS_1 contains $\overline{n}$ objects *closest* to the vantage point of A and S_2 contains the remaining ones. Remove all objects in S_2 from k-th child and reinsert them into (k+1)-th child. If the inserted object was in the set SS_1 insert it in the k-th child. At last update the boundary $A.\mu_k$:

$$A.\mu_k = \frac{\max(d(A.vp, s_i)\ \forall s_i \in SS_1) + \min(d(A.vp, s_i)\ \forall s_i \in SS_2)}{2} \tag{3.6}$$

3. Otherwise, we move objects from (k+1)-th inside k-th child (*expanding* k-th class). Collect objects from (k+1)-th child plus the inserted object and divide the objects into two subsets SS_1 and SS_2 so that SS_2 contains $\overline{n}$ objects *farthest* from the vantage point of A and SS_1 contains the remaining ones.Remove all objects in SS_1 from (k+1)-th child and reinsert them into k-th child. If the inserted object was in the set SS_2 insert it in the (k+1)-th child. At last update the boundary $A.\mu_k$ using formula 3.6.

Figure 3.2 shows the effect of the redistribution on a binary node. Pseudocode of the algorithm is in Appendix A.

The second operation mentioned above is node splitting. A node usually needs to be splitted if no redistribution is possible and the parent still has a place for one more child.

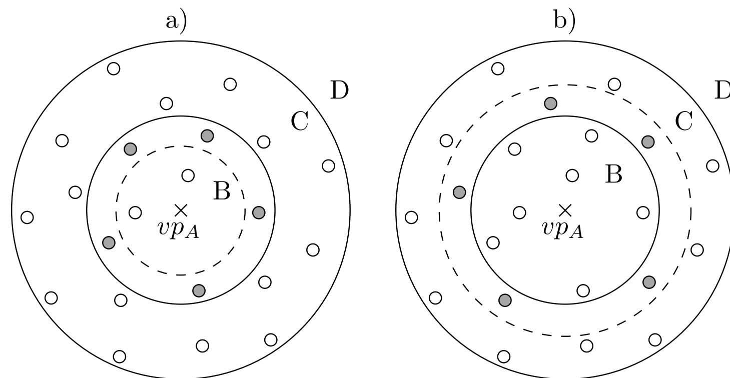

Figure 3.2: The effect of redistribution. Old boundaries are drawn in solid lines, the new boundary is dashed. Objects that are being moved are marked in grey. a) Shrinking class B by moving objects from B to C. b) Expanding class B by moving objects from C to B.

Otherwise the reorganisation would have to be propagated to the upper levels, involving even larger part of the tree. The task is to split the given node into two so that both new nodes contain part of the objects and additional free space. Unlike redistribution, split allocates new space in the tree. In [Fu et al. 2000] authors split either a leaf or non-leaf node using two algorithms whose semantic is, however, identical. Hence we only present a general method for splitting any node, be it a leaf or internal node. Let the splitted node be the k-th child of a node A. We collect all objects from the k-th child and order them by their distance to parent's vantage point. We divide the set into two subsets of equal sizes S_1 and S_2. We create a new node containing objects in S_2 and remove them from the k-th child node. We place the new node just after the k-the child as shown in Figure 3.3.

Having understood the meaning of redistribute and split operations, we now describe how to use them to reorganise the tree to provide free space needed to insert the new object while preserving the desired structure of the tree. To insert an object we traverse the tree starting at the root level. At each node, we measure the distance d from the new object to the vantage point and descend into a subtree, whose range covers d until we reach a leaf node L. If L has room for one more object, we insert the new object and the work is done. Otherwise we reorganise the tree in the following way:

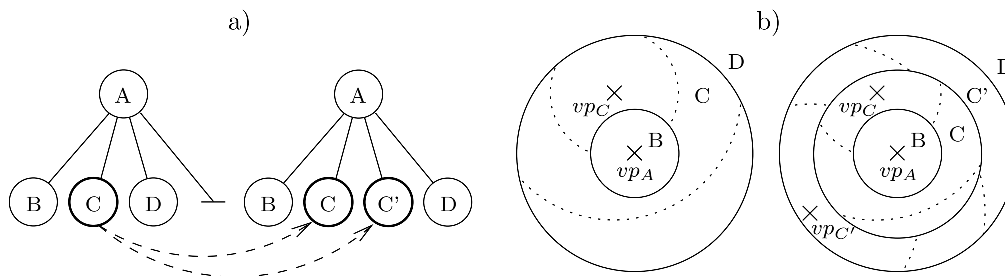

Figure 3.3: Splitting of node C. a) Splitting and inserting the new node in the tree. b) Updating the boundary. 2 levels of decomposition are shown for node C and C'.

1. Trace back looking for the nearest ancestor node A of L, for which $capacity(A) > count(A)$. Suppose that such an ancestor exists. Let B be the immediate child of A which is also ancestor of L.

   a) If A has a place for once more child, we split the node B and insert the new object in one of the newly created nodes, where appropriate.

   b) Otherwise, since $capacity(A) > count(A)$, there exist at least one sibling of B which can accommodate one more object. Find the nearest sibling of B and call it C. Redistribute the objects plus the inserted one amongst nodes B and C inclusively.

2. If no such node A is found or L is the root node, we split the root node. We create a new root with the old root as it's child and split it. The tree grows one level higher.

The strategy described above favours node splits over redistributions but the other way is also possible. Authors show experimentally that the split-first approach performs better in the terms of page accesses. The redistribute-first algorithm tends to keep the arity of the nodes low and does not allocate any more space in the tree. Therefore the redistributions are becoming more costly, until all children of the node reach saturation and the arity is increased by a split. Splitting as soon as possible reduces this effect by taking advantage of using as many child nodes as possible. Note that every now and then the process of

subsequent insertions reaches a state, where the whole tree is saturated and the root node has to be splitted. Using trivial split method described above, this means to rebuild the whole tree from scratch, which is very expensive. More sophisticated methods for node splitting can reuse the information about the underlying data set gathered up to the point of split. This is however, beyond the scope of this paper and is left for future work. Some ideas are sketched in Chapter 5.

3.2.2 Removal

When inserting objects, reorganisation of the tree is necessary to provide space for the inserted object by creating new nodes or moving objects from one part of the tree to another. To delete an object, no such reorganisation is crucial but for the sake of effectivity it is preferred to keep the tree in a compact form. In [Fu et al. 2000] it is achieved by eliminating underflown nodes. Authors define, that a node A underflows if it contains less objects than a threshold $MIN_{data}(A)$

$$MIN_{data}(A) = MIN_{leaf} \cdot (MIN_{fan})^{level(A)} \ , \tag{3.7}$$

where MIN_{leaf} is the minimum number of objects a leaf node should have, MIN_{fan} is the minimum number of children an internal node should have. If a node A underflows, it is *merged* with its neighbouring siblings if possible. Let the underflown node be the k-th child of its parent node A. We try to move objects from the underflown node to its adjacent siblings so that the underflown node can be removed from the tree. The merge algorithm is described below.

1. If (k+1)-th subtree can accommodate objects from k-th child: $capacity(A.child_{k+1}) \geq count(A.child_k) + count(A.child_{k+1})$, we move all objects from the k-th child to (k+1)-th and remove k-th child from the tree.

2. If (k-1)-th subtree can accommodate objects from k-th child: $capacity(A.child_{k-1}) \geq count(A.child_{k-1}) + count(A.child_k)$, we move all objects from the k-th child to (k-1)-th

and remove k-th child from the tree.

3. Otherwise, if both (k-1)-th and (k+1)-th subtrees can accommodate the objects from k-th child, i.e. $capacity(A.child_{k-1}) + capacity(A.child_{k+1}) \geq count(A.child_{k-1}) + count(A.child_k) + count(A.child_{k+1})$, we calculate how many objects should be moved from the k-th child to (k-1)-th:

$$n = \frac{count(A.child_{k-1}) + count(A.child_k) + count(A.child_{k+1})}{2} - count(A.child_{k-1})$$

We move n objects from the k-th child to (k-1)-th and the remaining objects to (k+1)-th subtree and remove k-th child from the tree.

If the objects from the underflown node cannot be accommodated in its adjacent siblings, we redistribute all objects from A among its child subtrees excluding k-th, which is removed at last. If A has only one child after merging, the child replaces A to reduce the tree height.

To remove an object from the tree, we traverse the tree in the same way as the insertion algorithm until we reach a leaf node L. We remove from L and if it does not underflow, the work is done. If L underflows, we perform one of the following steps:

1. Trace back looking for the nearest ancestor node A of L, for which $count(A) \geq MIN_{data}(A)$. Suppose that such an ancestor exists. Let B be the immediate child of A which is also ancestor of L.

   a) If one adjacent node or both adjacent nodes together can accommodate the objects is B, merge the node B. If A has only one child after merging, replace A with $A.child_0$.

   b) Otherwise, we redistribute all the objects under A among its child nodes excluding B and remove B from the tree.

2. No such node A is found or L is the root node. If the root node is an internal node we merge the child node from which the object was removed. If the root node only has one child after merge, the child becomes new root. If the root is a leaf node, hence L is the root node, no further action is taken.

3.3 Dynamic indexing using Hierarchy of Clusters

The algorithms presented in Section 3.2 can be adapted to work with Hierarchy of Clusters described in Chapter 2. We pinpoint two major differences between the indexes. At first, HC uses a function $h(n)$ to control the balancedness of the tree. For this reason we cannot rely on formula 3.3 and 3.7 to calculate capacity and underflow threshold for a node. We redefine the capacity of a node A as the number of leaf nodes under A multiplied by the bucket size. Note that the capacity can no longer be determined from position of A in the tree but relies merely on structure of its subtrees. That has an impact on the removal algorithm, as described later. The second difference is that Hierarchy of Clusters is strictly a binary tree. This fact simplifies implementation of the redistribute, split and merge methods, as well as the insertion and removal algorithms themselves. In a binary tree, two siblings are always adjacent, hence there is no need for the general redistribution and merge methods operating on non-adjacent subtrees. For simplicity we refer to $child_1$ as to the $left$ child and to $child_2$ as to the $right$ child. In the same manner we refer to the boundary between the left and right child of a node as to μ, meaning μ_0.

Inserting or removing single object is inefficient and not practical. In real applications we are often interested in updating many objects at a time. A naive solution would be to insert or remove all objects one by one in a loop. The disadvantage of this approach is that for an arbitrary node in the tree the reorganisation may be forced in every iteration of the loop. Consider insertion in the DVP-Tree. The probability that a node will require reorganisation grows with growing amount of objects in the tree, compared to its capacity. In extreme case, when the tree is nearly full, the probability approaches 1. Inserting objects will very likely reorganise large part of the tree in every iteration, until the saturation level is reached and a split is launched. This behaviour can be eliminated by inserting objects in blocks and forcing a reorganisation as soon as it is clear a node will overflow or underflow. The method we present in this section is suitable for both insertion and removal of objects and guarantees the reorganisation to happen once per node at maximum. To increase efficiency of single object insertions or removals, we can for example store the inserted or removed objects in an auxiliary index and delay modification of the main index until a certain amount of changes

had been requested and then execute all at once, saving lot of effort for reorganisation.

To assure correct structure of the HC tree and allow multiple object insertion, we modify the redistribution and split methods from the previous section. As for the redistribute operation, only two modifications are necessary. At first, we replace the mean in condition (3.5) with the $h(n)$ function so that each redistribution will assure the correct amount of objects in both nodes. To support multiple object insertion we give the algorithm two sets of inserted objects, one for each node. The size of the sets helps to calculate the right amount of objects to be moved so that the $h(n)$ criterion is met after insertion. The new redistribution algorithm which redistributes objects among both subtrees of a node A is shown below. O_1 and O_2 are the candidate objects to be inserted in the two subtrees. Pseudocode of the algorithm can be found in the appendix.

1. Calculate the amount of objects the left subtree should have.

$$n = h(count(A.left) + |O_1| + count(A.right) + |O_2|) \tag{3.8}$$

2. If $count(A.left)+|O_1| > n$ we move objects outside the left child. Let $S = [A.left] \cup O_1$. Divide S two subsets SS_1 and SS_2 so that SS_1 contains n objects closest to the vantage point of A and S_2 contains the remaining ones. Remove all objects in SS_2 from the left child and reinsert them into the right one. Then insert all objects present in both O_1 and SS_1 into the left child and all objects in O_2 into the right child. Update the boundary $A.\mu$ using (3.6).

3. Otherwise, we move objects from the right child inside the left. Let $S = [A.right] \cup O_2$. Divide S into two subsets SS_1 and SS_2 so that SS_2 contains n objects farthest from the vantage point of A and SS_1 contains the remaining ones. Remove all objects in SS_1 from the right child and reinsert them into the left one. Then insert all objects present in both O_2 and SS_2 into the right child and all objects in O_1 into the left child. Update the boundary $A.\mu$ using (3.6).

Note that the objects are inserted and removed all at once using the methods described

later. When removing objects from the left or right child, we prefer no reorganisation (redistributions or node merging) because the free space obtained will be used for inserting objects from O_1 or O_2.

DVPT allowed splitting of a node if its parent had a place for one more child to accommodate the second fraction of the splitted node. Such a situation does not occur in HC because the arity of internal nodes is fixed. Splitting has, however, importance for the insertion algorithm as it increases the tree capacity. We will use a trivial split operation that collects objects from the splitted node and builds a new tree using method for static HC construction. This is, however, very inefficient approach. A possible solution is to reuse the existing decomposition and is left as an open problem. The main obstacle is how to allocate a new space in such a tree and how to choose new pivots. Note that the new split forces the tree to grow in height rather than in width. To increase the performance of insertions, we force the split algorithm to create a tree which is not entirely full, so that the split operations will be less frequent. In [Fu et al. 2000] a reasonable value of *leaf utilization* was set to 70%.

The merge algorithm does not require any modification as its task is simply to concatenate objects from two nodes into a single one and hence does not depend on the way objects are distributed amongst the nodes. Since Hierarchy of Clusters is a binary tree, merge will, however only consider two adjacent siblings and will always reduce the height of the underlying tree because the number of children of the parent node was reduced to one. The parent is then replaced with the merged node.

3.3.1 Multiple object insertion

To insert a set of objects into the tree, we traverse the tree starting at the root. In every internal node we divide the inserted objects into two subsets to be inserted in the left and right subtrees. Let A be an internal node and O_1 and O_2 the candidate sets to be inserted in $A.left$ and $A.right$. We use the sizes of the two sets to immediately determine, whether a reorganisation is needed in a following way:

1. If $free(A.left) \geq |O_1|$ and $free(A.right) \geq |O_2|$ we continue recursively in both

subtrees.

2. Otherwise, if the overall free space can accommodate objects from O_1 and O_2, i.e. $free(A.left) + free(A.right) \geq |O_1| + |O_2|$, we redistribute the objects from $A.left$ and $A.right$ plus the inserted objects O_1 and O_2 among left and right child nodes of A.

3. Otherwise the node A overflows and has to be splitted.

If a leaf node is reached, the objects are stored and the work is done. It is interesting to remark that the leaf node usually does not overflow because such a situation would be handled in the higher levels of the tree by the steps 1 and 2. The only exception is the root node, which must be split if the overall free space in the tree is insufficient for storing the new objects. We expect a behaviour similar to the DVP-Tree, where the amount of work needed to insert new objects depends on the amount of free space in the tree.

3.3.2 Multiple object removal

Using similar approach as for the insertion, one might consider a node to underflow if the amount of objects stored in it is lower than its capacity, multiplied by a specified threshold value. While this is fine for the DVP-Tree, this approach cannot be used in HC because the correct capacity of a node is not known. Therefore we look for a different way to recognise an underflown node. We measure the error of an internal node A as the square of the relative difference of the number of objects under left (or right) child to from the ideal case.

$$E_N(A) = \left(\frac{h(count(A)) - count(A.left)}{h(count(A))} \right)^2 \tag{3.9}$$

In addition to this we define $\varepsilon, \varepsilon \in \langle 0, 1 \rangle$ as a maximal error allowed per node. If $E(A) > \varepsilon$, the node A has to be reorganized.

When inserting objects, we could immediately decide whether the node needs reorganisation from size of the inserted set, eliminating the effort for restructuring the tree to the

necessary minimum. For removal such a technique can only be used if all the removed objects are indeed present in the tree. This is not practical for situations when the amount of objects truly removed is not known beforehand. An example would be a range removal, where all the objects within a specified distance from the given objects shall be removed. For this reason we split the removal operation into two phases. In the first phase we remove the candidate objects the tree without any reorganisation. This can be achieved by traversing the tree, dividing the set of candidates into two sets according to their distance from the vantage point and the boundary value μ. When a leaf node is reached, all objects in the candidate sets are removed. In the second phase, we *pack* the tree into a compact form by eliminating all empty and underflown nodes. We traverse the tree starting at the root and in each internal node A we use the following strategy:

1. Calculate the square error $E_N(A)$ using formula (3.9).
2. If $E_N(A) > \varepsilon$, the left and right subtrees are not balanced enough hence require reorganisation.
   a) If $free(A.left) \geq count(A.right)$ or $free(A.right) \geq count(A.left)$, merge the left and right child and replace A with the result of the merge.
   b) Otherwise redistribute the objects among $A.left$ and $A.right$.

Changing the value of ε, one can optimise the removal speed as compared to the quality of the tree. Setting ε low will produce trees close to the optimal state but removing objects will require many reorganisations. Higher error, on the contrary, can be found useful if we are removing objects frequently and prefer speed over the quality.

3.3.3 Insertion with error control

The error function defined above can be useful for designing an error driven insertion method, where the necessity of reorganisation is decided by the quality rather than by insufficient capacity. The disadvantage of using capacity as the limit is that the amount of free space

will affect the overall quality of the tree. An almost full tree will tend to have node with low error while a tree with large capacity and very few objects may loose its qualities after massive insertions.

Like in the previous insertion method, we traverse the tree starting at the root. In every internal node A we divide the inserted objects into two subsets O_1 to be inserted in the left child and O_2 to be inserted in the right one. We use the following strategy to check whether the node needs reorganisation and take the appropriate action.

1. If the overall amount of free space under A is sufficient for inserting O_1 and O_2, that is $free(A.left) + free(A.right) \geq |O_1| + |O_2|$, we predict the error $E(A)$ as if the insertions were completed.

$$E_N(A) = \left(\frac{h(count(A)) - (count(A.left) + O_1)}{h(count(A))} \right)^2$$

   a) If $E_N(A) > \varepsilon$ we redistribute the objects from $A.left$ and $A.right$ plus the inserted objects O_1 and O_2 among left and right child nodes of A.

   b) Otherwise if each child can accommodate the inserted objects, we continue recursively in both children.

2. Otherwise the node A overflows and has to be splitted.

Even though the two algorithms look similar, there is a fundamental difference in how the capacity is being treated. The first insertion algorithm only allows node splitting at the root level, so that the insertion has to reorganise large part of the tree in order to provide place for the inserted objects. If this is not possible, the root node is splitted, increasing the tree height and capacity. The former method, however, allows a node to be splitted anywhere in the tree, which eliminates the need to reorganise large parts of the tree in search for available space. The use of the split and redistribute operations is controlled by the ε parameter.

We call the algorithm *exact insertion* to differentiate from the previous insertion algorithm. The name comes form the fact that the lower bound on the quality of the tree can be set explicitly by the parameter ε.

3.4 The tree quality

To express the quality of the tree quantitatively we measure a weighted average of the relative square error of all internal nodes. We prefer the weighted average to make the overall error relative to the tree size. This way the quality of two trees with different sizes can be compared. We define the average error of a tree T as

$$E_T = \frac{\sum_{A \in T}(E_N(A)^2 \cdot w_A)}{\sum_{A \in T}(w_A)} \ , \tag{3.10}$$

where w_A is the weight of node A

$$w_A = \frac{count(A)}{count(root)} \tag{3.11}$$

Since ε is the upper bound for the square error for every internal node, it is also the upper bound for the average error E_T of the tree.

$$E_T \leq \varepsilon \tag{3.12}$$

Note that E_T refers to the (un)balancedness of the tree, as compared to its ideal state where each node contains $h(n)$ objects in its left and $n-h(n)$ objects in its right child (where n is the amount of objects stored under the node). E_T alone, however, does not determine the quality of the partition and hence lower error does not ensure better performance of search operations. Among other factors influencing the partition quality we shall recall the pivot selection strategy, which can lead to major improvements.

3.5 Complexity analysis

We use amortized analysis to show that the insertion cost is proportional to the tree height. Our complexity measure will be the number of calculated distances. We define three opera-

tions: `traverse`, `redistribute` and `split`.

`Traverse` is an amortized operation of comparing an object against a node to decide which path to continue. For this we need to compare the object against the pivot, hence the cost of the operation is 1.

`Redistribute` is the operation of redistribution under a certain node. If n is the number of objects stored under the node then a single redistribution requires $O(n)$ distance calculations. In the worst case the redistribution may continue recursively until the leaf nodes are reached, hence requiring $O(kn)$ distances to be calculated, where k is the height of the tree rooted at the node.

`Split` is the operation of splitting of a node and its worst case complexity is $O(kn)$, where n is the number of objects under the node and k is the height of the new tree. This corresponds to rebuilding the whole tree from scratch.

Let us first assume a simplified case without redistributions, as if the objects were being inserted in such a way that no reorganization is needed until the tree is full and the root is splitted. We try to find such value of the amortized cost $C_{traverse}$ for the `traverse` operation so that it is able to pay off for the `split` operation triggered after a certain amount of traversals has been finished. We assign `split` the amortized cost of zero. Having an empty tree of capacity m with height k, we insert the first m objects in $O(mk)$ (we need k traversals to reach a leaf node). Each `traverse` operation donates a certain amount of *credits*, which equals the difference between the amortized and the real costs of the operation. Inserting $(m + 1)$-th object requires splitting of the tree before the actual insertion. If the newly constructed tree was of the same height, the split cost would be $O(km)$ and therefore $2km$ credits would be required (km to reinsert m objects from the original tree and km to reinsert another m objects from the original tree). Since the tree grows in height by one, the split cost will in fact be $O((k + 1)m)$ and we need extra m credits for the splitting. We cannot count on the first $m/2$ objects in the tree to pay for the new one, as it already paid for the current one, so we rely on the last $m/2$ objects to donate the necessary m credits. We also have to add the real cost of the operation. We can calculate the amortized cost for the

`traverse` operation as

$$C_{traverse}(k,m) = \frac{2km}{km} + \frac{m}{\frac{km}{2}} + 1 = 2 + \frac{2}{k} + 1.$$

Since the cost has to be a constant, we take the limit of the $\frac{2}{k}$ expression for $k \to \infty$.

$$C_{traverse}(k,m) = 3$$

For a large tree, the effect of increasing height becomes negligible and therefore no addition is in fact needed to the amortized cost of the `traverse` operation. Note that the above analysis assumes that the tree always doubles its capacity upon root split. That corresponds to the utilization ratio of 50%.

Redistributions do not occur in a way splits do. It is not possible to say when a redistribution will happen because it depends on the extent of utilization of individual leaf nodes. Let us consider the worst scenario when inserting an object triggers redistribution in the highest node, which subsequently forces the lower nodes to redistribute as well, resulting in redistribution in every node. Having a tree of height k which contains m objects, such a global redistribution would require $O(km)$ distance calculations. At this point there must be enough credits since every insertion donates $2k$. It is important, that the redistribution balances out the utilization of the affected branches. As a result, following redistribution will move at most one object at each node of the tree along the path of insertion (since all branches are balanced out and a single object is being inserted). Therefore the donation made by the inserted object will be sufficient for the redistribution. Under these circumstances we can say that the number of distance calculations caused by redistributions is not greater than $2k(m+1)$ for any tree of height k containing m objects. We increase the amortized cost of the `traverse` operation by 2 so as to pay off for the redistributions (1 for removing an object during redistribution and 1 for inserting it into new node).

$$C_{traverse}(k,m) = 5$$

Since `traverse` is the only operation with nonzero amortized cost and we perform k traversals in order to insert an object, we can see that the total amortized complexity of the insertion is proportional to $5k$ or

$$T \in O(k)\ . \tag{3.13}$$

Same principle applies to the removal algorithm.

Chapter 4

Experimental results

To study performance of the update algorithms, we implemented Hierarchy of Clusters and the proposed insert and remove methods in C++. The implementation is independent of the indexed data and the distance function as well as the $h(n)$ function and pivot selection strategy. Currently the whole index is kept in the primary memory and we use pointers to objects in nodes. The objects themselves are stored in a separate array. Due to this organization we did not experiment with the disk access cost of the tested operations but merely used the number of distance computations as a complexity measure. The two costs are, however, related since more distance computations require accessing more disk pages and vice versa so that reduction of I/O cost can be achieved by both decreasing the number of distance calculations and organizing the nodes effectively in a page. A disk organization similar to [Fu et al. 2000], where the nodes are packed into pages in breadth-first order is possible for Hierarchy of Clusters as well.

We used two data sets for testing, each containing 50,000 objects. The first data set consisted of 15 dimensional vectors picked randomly from uniform distribution on interval $\langle 0, 1 \rangle$ on each dimension. The indexes that operated on the dataset were strongly unbalanced using $h(n) = \sqrt{n}$. The second data set consisted of 10 dimensional vectors that formed clusters. The number of clusters was chosen to be 100 and their centers were picked at random but respecting that no object inside the cluster falls outside the unit hypercube.

Objects from every cluster were uniformly distributed on interval $\langle -0.1, 0.1\rangle$ relative to the cluster center. The clustered data set was considered rather low dimensional since its insintric dimensionality was expected to be less than the representational. Hence we used $h(n) = n/4$ for indexes that worked with the dataset. Pivots were picked at random from the input dataset as it does not require extra distance calculations and hence does not affect the overall cost of the tested algorithms.

4.1 Block size for updating multiple objects

First we tested how the amount of objects updated at once affects the performance. To test the insertion method, we created an empty index and inserted 50,000 vectors in blocks of 10, 100 and 1000 objects at once. For better results, we set the leaf utilization rate to 70% as suggested in [Fu et al. 2000]. The construction costs for both data sets are shown in Figure 4.1. We also show the cost of the static construction for comparison.

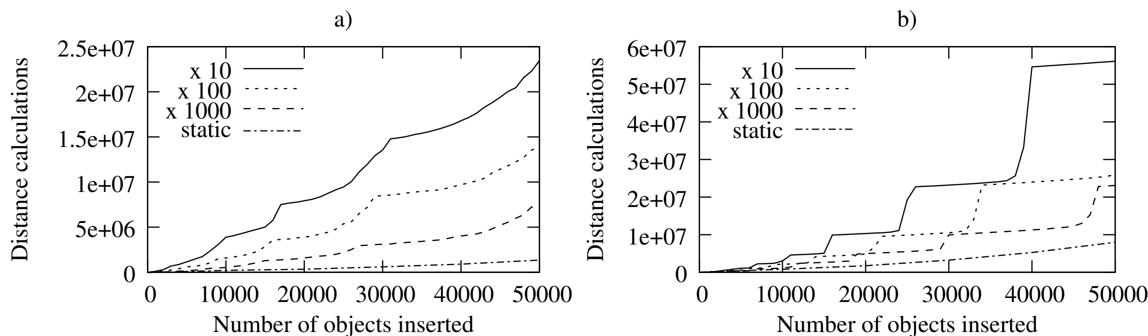

Figure 4.1: Insertion cost in distance calculations for blocks of 10, 100 and 1000 objects. a) 10 dimensional clustered data, $h(n) = 1/4$. b) 15 dimensional uniformly distributed data, $h(n) = \sqrt{n}$.

We can see that both cost functions have a characteristic pulsating shape. Regions of the plot, where the function is steepest correspond to the tree as it is approaching the saturation level. The closer, the larger part of the tree needs to be redistributed to provide enough place for the inserted objects, demanding more and more distance calculations. When the

tree cannot accommodate any more inserted objects, the root node is splitted and the whole tree is rebuilt. From that point on, the tree has enough of free space and can handle the coming insertions with little effort, until the free space is consumed and the cycle is repeated. Using blocks of higher sizes reduces the amount of executed redistributions, we observe the largest benefit when the tree is close to saturation since every redistribution is very costly. The efficiency of splitting of the root node is not affected. Note that the insertion cost grows rapidly as the tree is becomes more unbalanced. This is because unbalanced trees are taller than balanced ones, hence requiring more pivot–object comparisons when traversing the tree. The same problem is present while building the tree statically. The complexity of the static construction is $O(kn)$, which can turn out to be $O(n \log n)$ for a balanced tree or $O(n^2)$ in case of an extremely unbalanced one.

We carried out identical test for the exact insertion method with fixed value of $\varepsilon, \varepsilon = 0.01$. The benefit of inserting larger amount of objects at once was not as high as in the previous case but still notable. The reason for it is that the exact insertion controls reorganizations by the parameter ε. Hence the tree has no saturation level and cost for redistributions does not dominate the overall construction cost anymore. Figure 4.2 shows the result for the 15 dimensional space. The advantage of exact insertion is that it has more predictable behaviour because of the absence of the saturation barrier. Additionally, the speed of insertion can be regulated by setting the error threshold ε.

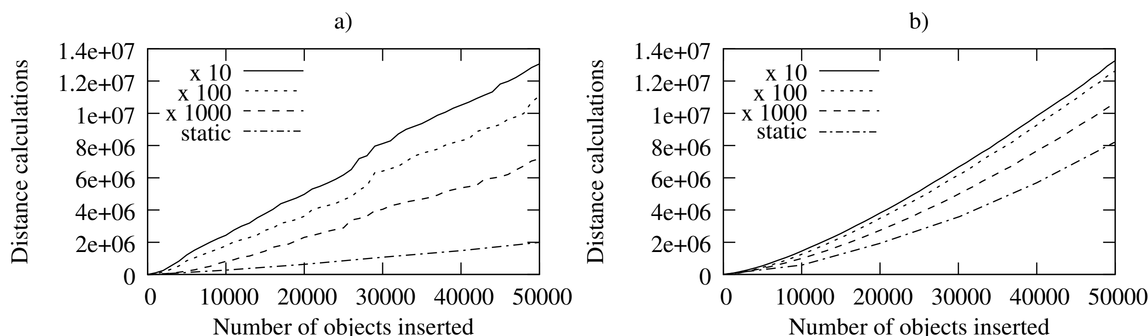

Figure 4.2: Insertion cost in distance calculations for exact insertion with $\varepsilon = 0.01$ and blocks of 10, 100 and 1000 objects. a) 10 dimensional clustered data, $h(n) = 1/4$. b) 15 dimensional uniformly distributed data, $h(n) = \sqrt{n}$.

At last we put the removal algorithm to test. We built an optimal HC index statically on the whole data set and then removed all 50,000 vectors in blocks of 10, 100 and 1000. We set the leaf node utilization ratio to 100% to achieve faster removal. In practice, however, lower value might be preferred as the insertions and removals are combined. Results for both data sets are shown in Figure 4.3.

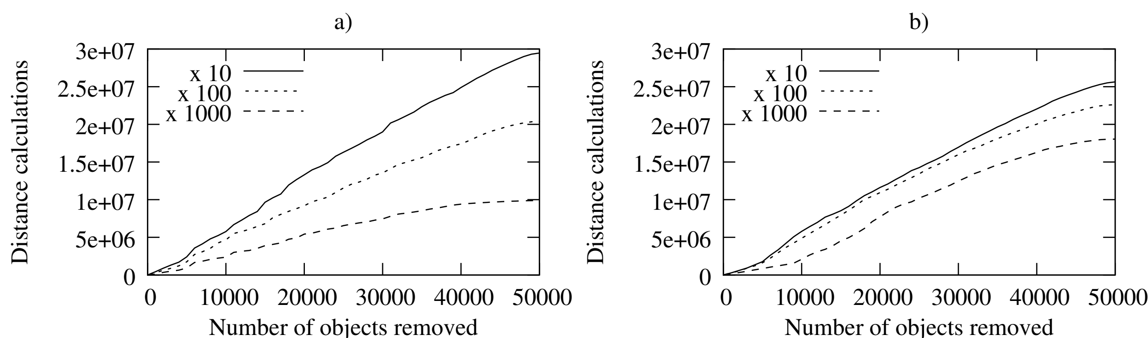

Figure 4.3: Removal cost in distance calculations for $\varepsilon = 0.01$ and blocks of 10, 100 and 1000 objects. a) 10 dimensional clustered data, $h(n) = 1/4$. b) 15 dimensional uniformly distributed data, $h(n) = \sqrt{n}$.

4.2 Mean square error

Besides the construction time, we also focused on the quality of the trees after a large amount of objects has been inserted or removed. We used the same data sets as above and conducted several tests to study the behaviour of the exact insertion and removal algorithms for varying ε. Like in the previous section, we created an empty index and inserted 50,000 objects using the exact insertion algorithm. We tried $\varepsilon = 0.01$, 0.05 and 0.1, the size of block was fixed to 1000 objects. Figure 4.4 shows the mean square error E_T (see Section 3.4) for both data sets and chosen ε. We included the error and construction cost of the non exact insertion method for comparison.

From plots 4.4a and 4.4b we can see that the exact insertion method introduces lower error than the non exact insertion algorithm. The construction cost is more predictable

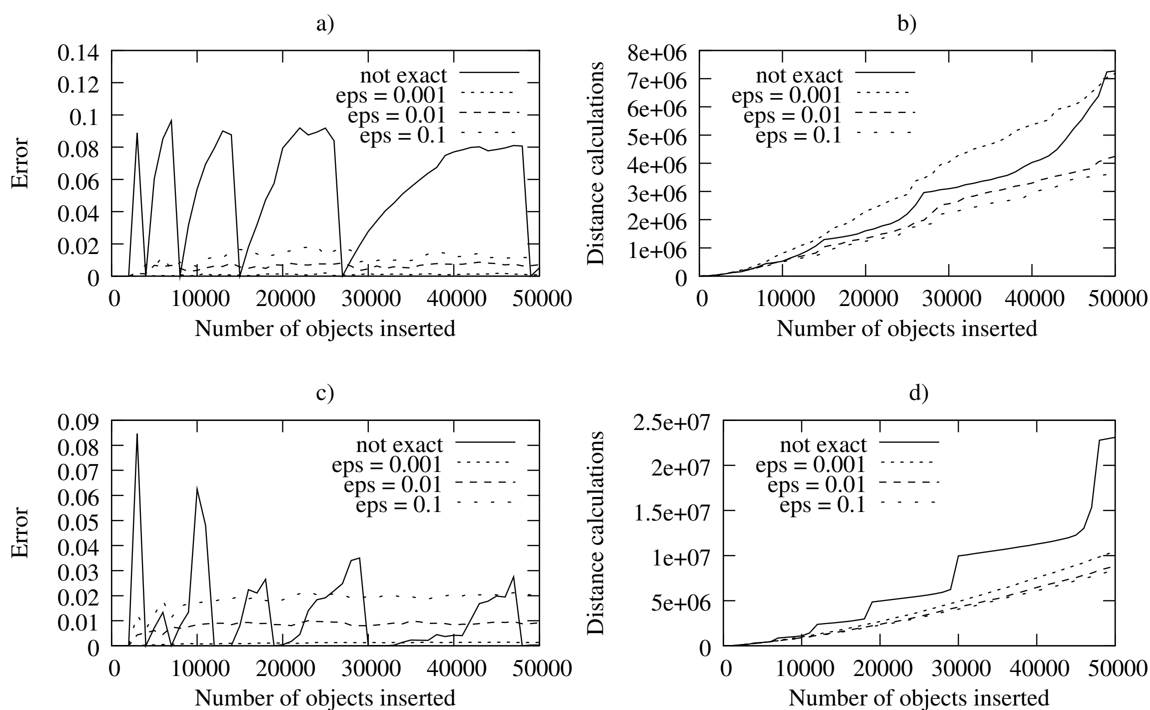

Figure 4.4: Error and construction cost for exact insertion for $\varepsilon = 0.01$, 0.05, 0.1. At the top 10 dimensional clustered vector space, at the bottom 15 dimensional space of uniformly distributed vectors.

because there is no saturation barrier after which a global reorganization would be needed. Note that the error function for the non exact insertion has a pulsating character closely related to its construction cost. When the tree is nearly full, the little amount of free space only allows a minimal error. As the tree is approaching the saturation level, the allowed error is very small and therefore every insertion is very expensive. When splitting the root node, the tree is rebuilt with optimal amount of objects under every node, hence zero error. Plots 4.4c and 4.4d show similar error for the removal algorithm.

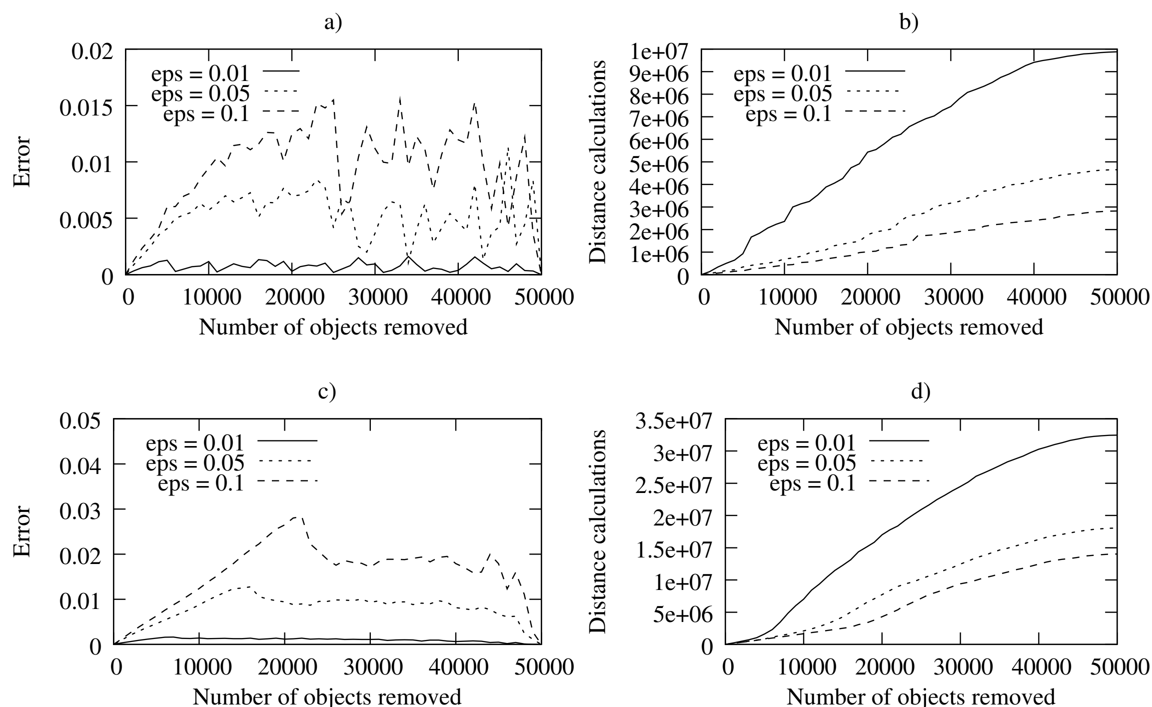

Figure 4.5: Error and construction cost for object removal for different values of ε = 0.01, 0.05, 0.1. At the top 10 dimensional clustered vector space, at the bottom 15 dimensional space of uniformly distributed vectors.

Note that for both algorithms the error converges to a value that is much lower than ε. This is caused by frequent redistributions and splits, which restore correct (un)balance in the affected portion of index. Therefore the error reaches the maximum value very seldom. A probabilistic solution might find a higher value of ε, $\varepsilon' > \varepsilon$, for which the actual E_T would

still be within ε with some confidence level.

4.3 Search comparison

To evaluate the quality of the index in practical way, we tested the search performance on both data sets, with trees of different error. We used the exact insertion method and inserted 50,000 objects into an empty index with error thresholds $\varepsilon = 0.01$, 0.05, 0.1 and using the non-exact method. Then we run a series of range queries with increasing radius and measured the number of distance calculations needed to find the answer. To eliminate random error, we run every range search with 100 different queries and calculated the average number of distance calculations. For both datasets, the range was chosen so that 0.2% of the database was retrieved at maximum. We used the pivot selection strategy that maximizes the sum of distances of the pivot to the previously selected ones as proposed in [Brin 1995, Chavez and Navarro 2000, Chávez et al. 2001b]. To increase the performance, the selection did not operate on the whole dataset but 100 random candidates were chosen. Similarly, since the number of pivots may be high for unbalanced trees, the algorithm picked 50 random pivots from the previously selected ones to compare against the candidate set. Results are shown in Figure 4.6.

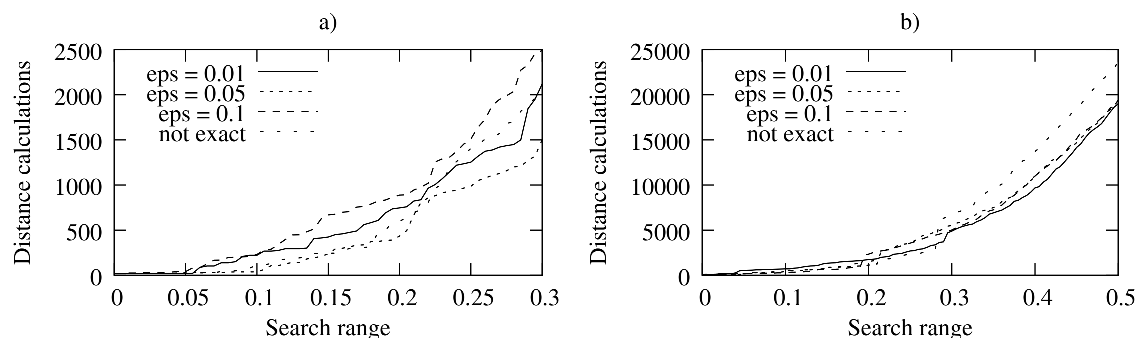

Figure 4.6: Search efficiency for different values of $\varepsilon = 0.01$, 0.05, 0.1. and for non-exact insertion method. a) 10 dimensional clustered vector space, b) 15 dimensional space of uniformly distributed vectors.

From both plots we can see that the impact of the error on the search is minimal. As mentioned in Section 3.4, the error E_T serves rather as way to quantify the (un)balancedness of the tree, as compared to the ideal one, rather than determining the quality of the decomposition itself. The experiments show that the tree with lowest E_T does not necessarily yield best search performance, which is demonstrated in plot 4.6a.

Chapter 5

Conclusions and future work

We studied the problematics of metric indexing and various techniques that use the distance information to reduce the candidate set while searching. We surveyed existing metric indexes and payed special attention to how can they handle non-static data.

We studied in detail the algorithms for dynamic indexing proposed for Vantage Point Tree in [Fu et al. 2000] and adapted the algorithms for Hierarchy of Clusters. We extended both insertion and removal algorithms to operate on multiple objects. We show experimentally that inserting or removing multiple objects at once effectively reduces the amount of distance calculations. For the insertion algorithm, the number of distance calculations dropped by nearly 60% inserting 1000 objects at once compared to inserting 10 objects at once. Moreover we proposed a new algorithm for insertion that uses a threshold parameter ε to limit the maximum allowed error. Exact insertion, unlike the original algorithm, is more sensitive to the (un)balancedness of the tree. While the almost balanced tree with $h(n)$ yielded 50% decrease of distance calculations inserting blocks of 1000 objects as compared to 10, for highly unbalanced tree $h(n) = \sqrt{n}$ the effect was not so strong. The number of distance calculations dropped by 20%. Similar results were achieved with the removal algorithm.

The advantage of the exact insertion algorithm is that the trade-off between insertion speed and tree quality can be controlled using the maximal error threshold. This is very

useful especially for highly unbalanced indexes, where the construction cost is very high. Another advantage is its predictable behaviour, where the cost function is not disturbed by the presence of a saturation barrier. In our tests, the exact insertion always required less distance computations for highly unbalances trees, compared to the original insertion method.

The presented dynamic indexes clearly beat the static ones in situations when a relatively small amount of new objects is inserted into a large database. The amortized average case complexity was shown to be $t \in O(k)$, where k is the height of the tree. The naive approach would require $O(kn)$ distance calculations to rebuild the whole index from scratch.

Open problems and our suggestions for future work:

In the search and remove methods, some distance computations could be saved when traversing the tree using the distance of the inserted/removed objects to k previous vantage points on the traversal path. For some objects we can guess whether they will fall into the left or right subtree of the node without actually having to calculate the distance from the current vantage point.

The main disadvantage of the insertion algorithm is the periodical reorganisation of the entire tree when the capacity is exceeded and the root node must be splitted. The same problem applies to DVP-Tree. More sophisticated split algorithm would significantly increase the insertion. Since splitting can be implemented using range search, a possible solution would be to use nearest neighbour search on the existing tree. The effectivity of nn search would, however, be a problem when the dimensionality of the space is high and the gain would be minimal.

Another option would be a lazy split operation, that preserves the decomposition in low level nodes, and only reorganises nodes in higher levels, treating the low level nodes as "unbreakable" groups. This would reduce the number of objects involved in the operation, as each group could be represented by its center and covering radius. On the other hand, this technique would require the decomposition to be symmetric.

From the experimental results we see, that the exact insertion and removal algorithms

tend to create trees with error much lower than the threshold ε. This is caused by frequent split and redistribute operations, which balance the tree to the optimal state. A probabilistic framework may be used to estimate a higher value of ε so that the overall error of the tree is within the requested limit with a certain confidence.

Bibliography

R. A. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *CPM '94: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 198–212. Springer-Verlag, 1994. ISBN 3-540-58094-8.

R. E. Bellman. *Adaptive Control Processes*. Princeton University Press, Princeton, NJ, 1961.

J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975. ISSN 0001-0782.

J. L. Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, 5(4):333–340, 1979.

S. Berchtold, D. A. Keim, and H.Kriegel. The x-tree: an index structure for high-dimensional data. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 28–39, 1996.

T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 357–368. ACM Press, 1997. ISBN 0-89791-911-4.

S. Brin. Near neighbor search in large metric spaces. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 574–584. Morgan Kaufmann Publishers Inc., 1995. ISBN 1-55860-379-4.

W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, 1973. ISSN 0001-0782.

B. Bustos, G. Navarro, and E. Chávez. Pivot selection techniques for proximity searching in metric spaces. *Pattern Recogn. Lett.*, 24(14):2357–2366, 2003. ISSN 0167-8655.

E. Chávez, J. L. Marroquín, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools Appl.*, 14(2):113–135, 2001a. ISSN 1380-7501.

E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*. To appear.

E. Chavez and G. Navarro. An effective clustering algorithm to index high dimensional metric spaces. In *SPIRE '00: Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, pages 75–86. IEEE Computer Society, 2000. ISBN 0-7695-0746-8.

E. Chávez and G. Navarro. A probabilistic spell for the curse of dimensionality. In *ALENEX '01: Revised Papers from the Third International Workshop on Algorithm Engineering and Experimentation*, pages 147–160. Springer-Verlag, 2001. ISBN 3-540-42560-8.

E. Chávez and G. Navarro. Probabilistic proximity search: Fighting the curse of dimensionality in metric spaces. *Information Processing Letters*, 85:39–46, 2003.

E. Chávez, G. Navarro, R. Baeza-Yates, and J. Luis Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, 2001b. ISSN 0360-0300.

B. Chazelle. Computational geometry: a retrospective. In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 75–94. ACM Press, 1994. ISBN 0-89791-663-8.

P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 426–435, Athens, Greece, August 1997. Morgan Kaufmann Publishers, Inc.

E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 26(1): 64–69, 1970. ISSN 0001-0782.

C. Faloutsos and King-Ip Lin. Fastmap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 163–174. ACM Press, 1995. ISBN 0-89791-731-6.

K. Fredriksson. Exploiting distance coherence to speed up range queries in metric indexes. *Information Processing Letters*, 95(1):287–292, 2005.

Ada Wai-chee Fu, Polly Mei-shuen Chan, Yin-Ling Cheung, and Yiu Sang Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *The VLDB Journal*, 9(2):154–173, 2000. ISSN 1066-8888.

A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57. ACM Press, 1984. ISBN 0-89791-128-8.

J. B. Kruskal and M. Wish. *Multidimensional scaling*. SAGE publications, Beverly Hills, 1978.

M. L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recogn. Lett.*, 15(1):9–17, 1994. ISSN 0167-8655.

G. Navarro. Searching in metric spaces by spatial approximation. *The VLDB Journal*, 11 (1):28–46, 2002. ISSN 1066-8888.

G. Navarro and N. Reyes. Fully dynamic spatial approximation trees. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002)*, LNCS 2476, pages 254–270. Springer, 2002.

G. Navarro and N. Reyes. Improved deletions in dynamic spatial approximation trees. In *Proc. of the XXIII International Conference of the Chilean Computer Science Society (SCCC'03)*, pages 13–22. IEEE CS Press, 2003.

J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984. ISSN 0362-5915.

E. V. Ruiz. An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recogn. Lett.*, 4(3):145–157, 1986. ISSN 0167-8655.

J. Uhlmann. Satisfying general proximity/ similarity queries with metric trees. *Information Processing Letters*, 40:175–179, November 1991.

P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 311–321. Society for Industrial and Applied Mathematics, 1993. ISBN 0-89871-313-7.

Appendix A

Pseudocode

In this appendix we present algorithms for insertion and removal on Hierarchy of Clusters proposed in Section 3.3.

```
Redistribute (A:tree node, O_1, O_2:set of objects)
{
  n = h(count(A.left) + count(A.right))
  if count(A.left) + |O_1| > n then {
    S = [A.left] ∪ O_1
    order objects in S by their distance to A.vp
    SS_1 = {S_1, S_2, ..., S_n}
    SS_2 = {S_{n+1}, S_{n+2}, ..., S_{|S|}}

    Remove (A.left, SS_2 − O_1)
    Insert (A.left, O_1 − SS_2)
    Insert (A.right, O_2 ∪ SS_2)
    A.μ = (max(d(A.vp, s_i) ∀s_i ∈ SS_1) + min(d(A.vp, s_i) ∀s_i ∈ SS_2))/2
  }
  else {
    S = [A.left] ∪ O_2
    order objects in S by their distance to A.vp
    SS_1 = {S_1, S_2, ..., S_{|S|−n−1}}
    SS_2 = {S_{|S|−n}, S_{|S|−n+1}, ..., S_{|S|}}

    Remove (A.right, SS_1 − O_2)
    Insert (A.right, O_2 − SS_1)
    Insert (A.left, O_1 ∪ SS_1)
    A.μ = (max(d(A.vp, s_i) ∀s_i ∈ SS_1) + min(d(A.vp, s_i) ∀s_i ∈ SS_2))/2
  }
}
```

Figure A.1: The redistribution algorithm for HC-Tree. If the algorithm is called from within an `Insert_EXACT` function, it also calls `Insert_EXACT` instead of `Insert` and packs the node from which objects were removed to ensure the minimal error.

```
Split (A:tree node, O:set of objects)
{
  S = [A] ∪ O
  A = Build_Tree (S)
}
```

Figure A.2: The trivial split algorithm. The algorithm rebuilds the whole splitted tree, possibly choosing new pivots.

```
Insert (A:tree node, O:set of Objects) returns tree node
{
  if A is leaf then {
    store O in A
  }
  else if free(A.left) + free(A.right) ≥ |O| {
    divide O into two subsets O1 and O2 so that
    O1 = {x ∈ O: d(x, A.vp) <= A.μ}
    O2 = {x ∈ O: d(x, A.vp) > A.μ}

    if free(A.left) ≥ |O1| and free(A.right) ≥ |O2| then {
      A.left = Insert (A.left, O1)
      A.right = Insert (A.right, O2)
    }
    else
      Redistribute (A, O1, O2)
  }
  else
    A = Split (A, O)

  return A
}
```

Figure A.3: The bulk insertion algorithm for Hierarchy of Clusters. Insert may return a new node in case of splitting, so we start the insertion by $root$ = `Insert` ($root$, O).

```
Insert_EXACT (A:tree node, O:set of Objects, ε:float) returns tree node
{
  if A is leaf then {
    store O in A
  }
  else if free(A.left) + free(A.right) ≥ |O| {
    divide O into two subsets O_1 and O_2 so that
    O_1 = {x ∈ O: d(x, A.vp) <= A.μ}
    O_2 = {x ∈ O: d(x, A.vp) > A.μ}

    n = h(count(A) + |O|)
    n_ε = n · ε
    if count(A.left) + |O_1| ≥ n − n_ε or count(A.left) + |O_1| ≤ n + n_ε then {
    {
      A.left = Insert_EXACT (A.left, O_1)
      A.right = Insert_EXACT (A.right, O_2)
    }
    else
      Redistribute (A, O_1, O_2)
    }
    else
      return Split (A, O)

    return A
  }
```

Figure A.4: The exact insertion algorithm for Hierarchy of Clusters. The algorithm may return a new node in case of splitting, so we start the insertion by *root* = `Insert_EXACT` (*root*, O, ε).

```
Merge (A:tree node, ε:float)
{
  if free(A.left) < count(A.right) {
    SRC = [A.right]
    DST = [A.left]
  }
  else if free(A.right) < count(A.left) {
    SRC = [A.left]
    DST = [A.right]
  }
  else
    let SRC be that child of A that stores more objects and
    DST the other one

  S = [SRC]
  Insert_EXACT (DST, S, ε)
  replace A with DST
}
```

Figure A.5: The merge algorithm. Merges left and right subtrees of A and replaces A with the result.

```
Remove (A:tree node, O:  set of objects)
{
  if A is leaf then
    remove objects in O from A
  else {
    divide O into two subsets O_1 and O_2 so that
    O_1 = {x ∈ O: d(x, A.vp) <= A.μ}
    O_2 = {x ∈ O: d(x, A.vp) > A.μ}

    Remove (A.left, O_1)
    Remove (A.right, O_2)
  }
}
```

Figure A.6: Remove algorithm.

```
Pack (A:tree node, ε:float) returns tree node
{
  if A is leaf then return A
  else {
    n_opt = h(count(A.left) + count(A.right))

    if count(A.left) < n_opt · (1 − ε) or count(A.left) > n_opt · (1 + ε) then {
      if free(A.left) ≥ count(A.right) or free(A.right) ≥ count(A.left) then
        A = Merge (A)
      else
        Redistribute (A, ∅, ∅)
    }

    return Pack (A, ε)
  }
}
```

Figure A.7: Pack algorithm. Pack may return a new node if the root subtrees were merged. We start the algorithm by $root =$ `Pack` ($root$, ε).