Efficient Algorithms to Execute Complex Similarity Queries in RDBMS

Adriano S. Arantes, Marcos R. Vieira, Caetano Traina Jr., Agma J. M. Traina

Computer Science Department - ICMC
University of São Paulo at São Carlos - USP
Avenida do Trabalhador São Carlense, 400
13560-970 - São Carlos, SP - BRAZIL
{arantes, mrvieira, caetano, agma}@icmc.usp.br

Abstract

Search operations in large sets of complex objects usually rely on similarity-based criteria, due to the lack of other general properties that could be used to compare the objects, such as the total order relationship, or even the equality relationship between pairs of objects, commonly used with data in numeric or short texts domains. Therefore, similarity between objects is the core criterion to compare complex objects. There are two basic operators for similarity queries: Range Query and k-Nearest Neighbors Query. Much research has been done to develop effective algorithms to implement them as standalone operations.

However, algorithms to support these operators as parts of more complex expressions involving their composition were not developed yet. This paper presents two new algorithms specially designed to answer conjunctive and disjunctive operations involving the basic similarity criteria, providing also support for the manipulation of tie lists when the k-Nearest Neighbor query is involved. The new proposed algorithms were compared with the combinations of the basic algorithms, both in the sequential scan and in the Slim-tree metric access methods, measuring the number of disk accesses, the number of distance calculations, and wall-clock time. The experimental results show that the new algorithms have better performance than the composition of the two basic operators to answer complex similarity queries in all measured aspects, being up to 40 times faster than the composition of the basic algorithms. This is an essential point to enable the practical use of similarity operators in Relational Database Management Systems.

Keywords: Query processing, complex similarity queries, similarity search algorithms.

1 Introduction

The currently available Relational Database anagement Systems (RDBMS) were developed to manipulate data expressed as numeric or short textual attributes, considering the total ordering relationship among the elements of these data domains. However, the volume and types of data stored and manipulated in the RDBMS has increased continually, and now includes several other data types. The new data types, commonly called complex data, usually do not present the total ordering relationship. Therefore, the existing search operations and the traditional indexing structures used in RDBMS are not useful.

Regarding complex data domains, such as image, video, spatial references, genomic se-

quences, time series, and others, the **similarity** between pairs of elements is the most important property [13]. Therefore, a new class of queries based on the similarity between elements emerged as the more adequate to manipulate data in complex data domains, that are called **similarity queries**. Similarity queries require the existence of a dissimilarity function on the data domain, also called a **distance function** or simply a "metric" [9].

There are basically two types of similarity queries in metric domains: the range queries expressed by the Rq predicate and the k-nearest neighbor queries expressed by the $kNNq$ predicate [18]. A range query recovers stored objects that differ up to a given dissimilarity degree from the query center. An example of a range query on a data set of genomic sequences is the following: "*Choose the polypeptide chains which are dissimilar from the given chain p by up to 5 codons*". A k-nearest neighbor query recovers the k stored objects that are the nearest to the query central object, where k is an integer value determining the number of objects retrieved. An example of a k-nearest neighbor query on the genomic data set is the following: "*Choose the 10 polypeptide chains nearest to the given polypeptide chain p*".

Most of the existing reports in the literature deal with the two similarity predicates implemented as isolated operations, not considering them as part of more complex expressions involving more predicates. In other words, existing algorithms designed to answer each one of these similarity queries do not allow optimizations that could be performed on combinations of them. Consequently, a complex similarity query involving more than one similarity operation tends to be processed inefficiently, requiring the execution of set-theoretical operators (as union and/or intersection) to combine the intermediate results obtained by the basic similarity operators.

The expansion of multimedia data stored in today RDBMS fosters the need of efficient ways to answer advanced queries, such as the similarity queries. A natural way to provide support to these data types is including support for similarity queries in the standard query language (SQL), allowing similarity predicates to be expressed as an extension of SQL. Hence, these predicates could be used as selection clauses together with the other existing clauses in SQL. To this intent, two main points need to be considered: how these predicates can be used together with others; and how operations composed of the basic predicates often used together can be supported by specific algorithms that are more efficient than the sequential execution of the basic algorithms followed by the set-theoretical operations.

In this paper we address the problem of how to develop specific algorithms combining similarity queries into more complex expressions and how to provide support for similarity queries in RDBMS. We propose two new algorithms, called $kAndRange()$ and $kOrRange()$, which provide specific support for complex similarity queries using the **AND/OR** clauses to compose queries centered at the same query object. Both algorithms are independent of the indexing structure used. To evaluate the effectiveness of the new algorithms they were implemented using both the sequential scan as well as the Slim-tree [23, 24] metric access method. Experimental measurements were performed comparing them with the measurements obtained by the execution of the two basic algorithms (range and k-nearest neighbors) followed by the required set-theoretical operations (the intersection operator to obtain the "and" operation, and the union operator to obtain the "or" operation). The results show that the proposed algorithms are more efficient and present better scalability, being able to reduce the number of disk accesses to as low as 1/12, the number of distance calculations to as low as 1/20, and executing more than one hundred times faster than the sequential execution of the basic algorithms followed by the set-theoretical operations.

A preliminary version of this paper was pre-

sented at SBBD 2003 [2]. Here, we show the following aspects that were not addressed in that previous version. We detail the treatment of tie lists in the k-nearest neighbor queries and how it affects the performance and the usability of the algorithms that perform k-nearest neighbor queries. We also provide examples of ties in complex queries involving k-NN queries in real data sets. And finally we detail the two proposed algorithms, and present a more complete evaluation of them, through the use of other two real and a synthetic data sets, including scalability experiments.

The remainder of this paper is structured as follows. In the next section, we first present a brief history of the development of algorithms to answer similarity queries. Section 3 presents required concepts and the motivation to develop the new algorithms. Section 5 presents the new algorithms $kAndRange()$ and $kOrRange()$. Section 6 describes the experimental results. Finally, Section 7 gives the conclusions of this paper.

2 Related Work

In the last years, algorithms to answer similarity queries have motivated many researches, most of them based on supporting hierarchical index structures. A common approach used is the "branch-and-bound" technique, where a tree is traversed from the root down to the leaf nodes. At each node, heuristics are used to determine which branches should be traversed next, and which branches can be pruned from the search. Pruning branches during the search requires to consider specific properties of the data domain.

One of the most influential algorithms in this category was proposed by Roussopoulos et al. [22], which finds the k nearest neighbors using an R-tree [14] to index points in a multidimensional space. Cheung and Fu [8] simplified this algorithm by reducing some heuristics while maintaining its efficiency. The algorithm proposed in [4] finds the nearest-neighbors of points continuously moving in a surface, also based on the work of Roussopoulos et al., and is the first one to consider multiple execution of the basic algorithms to perform complex similarity searches. The algorithms to answer similarity queries in metric spaces also follow the branch-and-bound approach, as those proposed to work on the M-tree [10] and on the Slim-tree [24]. Modifying the index structures to enhance branch-and-bound algorithms have also been considered, as for examples those proposed in the SS-tree [25] and in the SR-tree [17].

Other approaches were also proposed. One of them uses incremental algorithms to answer k-NN queries. A successful algorithm was proposed by Hjaltason and Samet [16]. It can efficiently find the $k + 1$ nearest neighbor after having find the k nearest neighbors. Park and Kim [21] proposed a complementary algorithm that can partially prune worthless tuples that will not fulfill the remaining non-similarity-based predicates in a query. The technique proposed by Hibino and Rundensteiner [15] processes incremental range queries in a direct manipulation through a visual query environment. An alternative proposed by Berchtold et al. [5] indexes an approximation of the Voronoi diagram associated to the data set. All of these works refer to algorithms considering just one simple similarity predicate.

More recently, operators to answer complex similarity queries combining more than one basic similarity predicate has been highlighted. Complex similarity queries over multiple features are considered in [6, 7, 12], and over a single feature in [11]. In [7, 12], the basic idea is that the evaluation of complex similarity predicates cannot be performed independently since the whole query depends on the combined scores of each single predicate. In [6, 11], index structures are used to enhance complex similarity queries in relevance feedback environments. Every algorithm in these works use an intermediary scoring function that, applied to each object retrieved by each basic predicate, evaluates the overall scores in or-

der to determine the answer of the complex similarity query. In all of these works, the operators were designed to be called explicitly and alone in a query command, as a predefined query. To the best of the authors' knowledge, no algorithm has been published aiming at combining similarity-based predicates into generic expressions.

Queries involving multiple similarity-based predicates are useful in many applications, and their combinations yield optimizations that can improve the performance of search operations. Therefore, the objective of this work is to provide algorithms that can be used to execute complex similarity queries, allowing optimizations to be detected and handled by the query optimization module of the RDBMS. The algorithms were designed to allow for algebraic rules to guide the query optimization process following the relational algebra. According to the best of the authors' knowledge, no other published work has achieved this goal before.

3 Motivation and Background

This section presents the fundamental concepts required to understand the proposed $kAndRange()$ and $kOrRange()$ algorithms, which are detailed in Section 5, and also the motivation for their development.

3.1 Metric Domains

Similarity queries can be posed only over data in a metric space. A metric space is a pair $\mathbb{M} = <\mathbb{S}, d()>$, where $\mathbb{S}$ denotes the universe of valid elements and $d()$ is a function $d : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{R}^+$ that expresses a measure of "distance"(dissimilarity) between elements of $\mathbb{S}$, that is, the smaller the distance, the more similar or closer are the elements.

A distance function must satisfy the following three rules to fulfill a metric space: **symmetry**: $d(s_1, s_2) = d(s_2, s_1)$, **non negativity**: $0 < d(s_1, s_2) < \infty$ if $s_1 \neq s_2$ and $d(s_1, s_1) = 0$, and **triangular inequality**: $d(s_1, s_3) \leq d(s_1, s_2) + d(s_2, s_3)$, where $s_1, s_2, s_3 \in \mathbb{S}$.

3.2 Similarity Queries

There are two main types of basic similarity predicates in metric domains. Considering a data set $S \subset \mathbb{S}$, these queries can be described as:

1. **Range Query -** Rq: given an object $s_q \in \mathbb{S}$ and a maximum search distance r_q, the range query represented by the $\sigma_{\left(R_q(s_q,r_q)\right)} S$ predicate selects every element $s_i \in S$ such that $d(s_i, s_q) \leq r_q$, that is:

$$\sigma_{\left(R_q(s_q,r_q)\right)} S = A_{R_q},$$
$$A_{R_q} = \{s_i | s_i \in S, d(s_i, s_q) \leq r_q\} \tag{1}$$

2. k**-Nearest Neighbor Query -** $kNNq$: given an object $s_q \in \mathbb{S}$ and an integer value $k \geq 1$, the k-nearest neighbor query represented by the $\sigma_{\left(kNNq(s_q,k)\right)} S$ predicate, selects the k elements $s_i \in S$ that have the shortest distance from s_q, that is:

$$\sigma_{\left(kNNq(s_q,k)\right)} S = A_{kNN},$$
$$A_{kNN} = \{s_i | s_i \in S, |A_{kNN}| = k,$$
$$\forall s_j \in S - A_{kNN} \Rightarrow d(s_q, s_i) \leq d(s_q, s_j)\} \tag{2}$$

Notice that the query center $s_q \in \mathbb{S}$ does not need to pertain to the data set S.

Access methods specific to index data in metric spaces, such as the M-tree [10] and the Slim-tree [23], are called Metric Access Methods (MAM). These structures were developed to improve the search algorithms that executes the similarity predicates. Efficient searching algorithms are important issues when retrieving multimedia data, as the cost of distance calculations on multimedia data is very high.

The fundamental property allowing similarity searching optimization is the triangular inequality. When the data set is indexed by a tree-based metric access method, this property enables the searching algorithm to prune whole

branches (subtrees), thus reducing the number of distance calculations needed to answer a query. As a consequence, better performance in the select operations is reached, because it is not necessary to compute the distances from the query center object s_q to every stored object s_i. The triangular inequality is able to perform branch pruning when one of the two following conditions holds [10].

$$d(s_{rep}, s_q) > r_{rep} + r_q, \quad (3)$$

$$d(s_{rep}, s_q) < |r_{rep} - r_q| . \quad (4)$$

where s_q is the query center, s_{rep} is the routing object in any intermediary node of the tree, r_q is the query radius and r_{rep} is the minimal covering radius of the node (or of the subtree). Table 1 shows the main symbols used in this paper.

Table 1: Table of symbols used in the paper.

$\mathbb{S}$	Set of all valid elements in the data domain.
S	Data set where queries are posed. $S \subseteq \mathbb{S}$
$d(s_1, s_2)$	Distance function, or dissimilarity function. $d : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{R}^+$, $s_1, s_2 \in \mathbb{S}$
d_k	The Dynamic radius in a $Nearest$ operator
k	Number of neighbors in a $Nearest$ operator
r_q	Range radius for $Range$ query
s_p	Routing object of a node
s_i	Object $\in S$
s_q	Query object (query center). $s_q \in \mathbb{S}$
s_{rep}	Routing object covering a subtree in a routing node
r_{rep}	Covering radius of the s_{rep} object in a node
tie	a variable indicating whether a tie list is required, or a sampling or a biased subset

3.3 Motivation

The main motivation to develop specific algorithms to answer complex similarity queries is that real systems often need the composition of similarity predicates, as exemplified in the following queries.

1. "*Choose at least 20 DNA sequences that are the most similar to the given sequence s including everyone differing up to 10 codons*":

   $\sigma_{kNNq(20,s)}$DNAdb $\cup$ $\sigma_{Rq(10,s)}$DNAdb;
2. "*Using a word processor, when a wrong word is written, show up to 10 words that differ at most 2 characters from the wrong word w*":

   $\sigma_{kNNq(10,w)}$WordDb $\cap$ $\sigma_{Rq(2,w)}$WordDb;
3. "*Find the 10 nearest restaurants from here that are not farther than 1 kilometer*":

   $\sigma_{kNNq(10,here)}$RestDb $\cap$ $\sigma_{Rq(1km,here)}$RestDb;

Although queries like these are common, existing algorithms deal only with the basic similarity queries. Moreover, the SQL standard does not include specifications for selection criteria based on similarity. However, there is currently a trend in support them, including the development of a standard to handle spatial data including similarity queries, as part of the ISO SQL/MM (SQL Multimedia Spatial Standard) [1, 19].

Currently, there exist some systems that support query commands involving similarity predicates on a limited basis. An example is the CIRCE system [3], aiming at extending SQL to answer similarity predicates on image data sets. However, multiple similarity criteria must be expressed in separated commands (through subselect commands) using the basic $Range()$ and $Nearest()$ algorithms to process the similarity predicates, combining their results using the set-theoretical operations.

However, this approach does not lead to the best performance regarding the execution time, the numbers of disk accesses and distance calculations. A more efficient approach should be to use algorithms tailored to answer complex similarity queries. This approach is similar to the one taken to implement the relational join operator. This operator is equivalent to the combination of two basic operations of the relational algebra, the Cartesian product followed by a se-

lection, although the join operator is much more efficient. In the same way, algorithms tailored to answer complex similarity queries would combine the basic algorithms into more sophisticated similarity operators, which can deliver an improved query answering procedure for similarity queries, allowing flexibility in its parameterization by the query optimizer of a RDBMS.

Therefore, it is important to develop algorithms that execute often-used complex similarity queries in a much more efficient way instead of the sequential execution of the basic algorithms. This is a required step in extending commercial systems to support complex similarity queries. This work proposes two new algorithms to execute conjunctions and disjunctions of similarity predicates, two of the most frequently used combinations of the basic algorithms. Table 2 summarizes the correspondence in the execution of the proposed algorithms and the combinations of the basic algorithms to answer queries.

4 Basic Algorithms for Similarity Predicates

This section discusses the basic $Range()$ and $Nearest()$ algorithms which execute the two main types of similarity predicates presented in Section 3.2. The specific case of ties in the $Nearest()$ algorithm is treated in Section 4.1.

The range query algorithm $Range(s_q, r_q)$ searches the data set S for the elements that are at distance r_q from the query center s_q or closer. The $Nearest(s_q, k)$ algorithm collects the k elements s_i that are the nearest in data set S to the query center s_q, sorted by the distance from each element s_i to the query center. The algorithm starts computing the distance from s_q to any element in S, until k elements are found, initializing the answer set with those elements. Afterward, a "dynamic radius" keeps track of the largest distance from elements s_i to s_q. Whenever an element s_i nearer to s_q is found, it replaces the farthest one in the answer set, reducing the dynamic radius accordingly.

This description applies both to searching through sequential scan as well as using an indexing structure. In the absence of an indexing structure, both algorithms require comparing the query center with every object stored in the data set. Due to the high computational cost to calculate the distance between pairs of elements in metric domains, similarity queries commonly use indexing structures to accelerate the processing, since they allow reducing the number of distance calculations by pruning subtrees. Consequently, index structures are even more important in metric domains than they are in domains that possess the total ordering property (the typical domains of the data handled in current RDBMS). However, sometimes an indexing structure does not exist, as for example when processing the intermediary results from previous selection operations, when creating an indexing structure is worthless. Sequential scans can be used in any situation, even when there is no indexing structure, so it is important the algorithms be able to be executed also through sequential scanning.

An index structure recursively groups objects under covering radii centered at representative objects, so the triangular inequality property can prune subtrees using a limiting radius and equations 3 and 4. The limiting radius in the $Range(s_q, r_q)$ algorithm is the range radius r_q, thus the pruning ability ("prunability") of this algorithm using index structures is usually high. As there is no static limiting radius to perform a k-nearest neighbor query, the dynamic radius is used as the limiting radius in the $Nearest(s_q, k)$ algorithm. Hence, until k elements are found, no pruning can be executed, and after that the dynamic radius may shrink, allowing that many unsuitable elements had been temporarily included in the answer set. Therefore, even using index structures, the $Nearest()$ algorithm tends to have a much lower prunability than the $Range()$ algorithm, which makes the cost of a nearest neighbor query larger than the cost of a range query, in general by one or two orders of magnitude.

Table 2: Equivalence of the proposed operators and the basic algorithms to answer conjunctive and disjunctive predicates.

	Complex Operator	**Answer-set composition of the basic algorithms**
Conjunction	$\sigma_{(Rq(s_q,r_q)\ \wedge\ kNN_tq(s_q,k,tie))}\ S$	$\sigma_{(Rq(s_q,r_q))}\ S \cap \sigma_{(kNN_tq(s_q,k,tie))}\ S$
Disjunction	$\sigma_{(Rq(s_q,r_q)\ \vee\ kNN_tq(s_q,k,tie))}\ S$	$\sigma_{(Rq(s_q,r_q))}\ S \cup \sigma_{(kNN_tq(s_q,k,tie))}\ S$

4.1 Tie Lists in $Nearest()$ Algorithms

The Levenshtein metric $L_{Edit}(s_1, s_2)$, also called the edit-string distance L_{Edit}, is a metric that counts the minimal number of symbols needed to be inserted, deleted, or substituted to transform the string s_1 into the string s_2. For example, $L_{Edit}($ "computer", "competent"$)$=3: two substitutions and one insertion. Searching an English dictionary with 25,153 words for the words differing up to two edit-string operations from the word "computer"has found 7 words: "computer", "compute", "copter", "compacter", "compote", "compete" and "commute", where the distance from "computer" to the first word is zero, to the second is one, and to the others is two. If a k-nearest neighbor query with $k = 3$ is posed, that is,

$$\sigma_{(kNNq(\text{"computer"},3))}\ EnglishWords$$

then there are five distinct correct answers. How a $Nearest()$ algorithm would treat this query? What elements should be returned?

In a first approach, the $Nearest()$ algorithm returns just k elements, including the objects that are nearer to the query center than the largest radius found, plus enough objects tied at the largest radius to complete the required quantity k. This approach returns a non repeatable answer to the query as posing the same query twice can bring different answers. However, it respects the required number k of elements in the answer. A second approach is to return the answer in two sets: the basic list L_b containing the objects that are nearer to the query center than the largest radius found, and a tie list L_t containing all the objects found at the largest radius distance of the query center. The answer of this approach is repeatable, but the application receives more than the number k of elements asked. When a tie list is required, the expression governing the k-nearest neighbor queries must be redefined as:

k-Nearest Neighbor query with tie list - kNN_tq: given an object $s_q \in \mathbb{S}$ and an integer value $k \geq 1$, the k-nearest neighbor query with tie list $\sigma_{(kNN_tq(s_q,k))} S$ selects at least k elements $s_i \in S$ that have the shortest distance from s_q such that:

$$\begin{aligned} \sigma_{\left(kNN_tq(s_q,k)\right)} S = A_{kNN} = L_b \cup L_t, \\ |L_b| \leq k, |L_b \cup L_t| \geq k, \quad \text{where} \\ L_b = \{s_i | s_i \in S, \\ \forall s_j \in S - L_b \Rightarrow d(s_q, s_j) > d(s_q, s_i)\}, \\ L_t = \{s_g, s_h | s_g, s_h \in S, d(s_g, s_q) = d(s_h, s_q), \\ \forall s_i \in L_b \Rightarrow d(s_q, s_i) < d(s_q, s_g), \\ \forall s_j \in S - \{L_b \cup L_t\} \Rightarrow d(s_q, s_j) > d(s_q, s_g)\} \end{aligned} \tag{5}$$

The basic $Nearest(s_q, k)$ algorithm can be changed to support answering k-nearest neighbor queries with tie lists by including a parameter tie, which indicates whether tie list should be returned or not. Thus, from now on we use the syntax of the $Nearest()$ algorithm as $Nearest(s_q, k, tie)$, where $tie = true$ means that a tie list must be returned in the answer set, otherwise, ties are arbitrarily chosen to return k elements.

Thus, rewriting the previous query in this section to

$$\sigma_{(kNN_tq(\text{"computer"},3))}\ EnglishWords$$

the answer set for the query is: $L_b \cup L_t$, where $L_b = \{$ "computer", "compute"$\}$ and $L_t = \{$ "copter", "compacter", "compote", "compete", "commute"'$\}$.

We consider that there are two basic approaches that a tie list-enabled $Nearest()$ algorithm can use to choose elements in the L_t list to return k elements when tie-lists are not requested, that we call *biased* and *sampled* tie lists. Each approach changes the way the $Nearest()$ algorithm chooses elements of the internal L_t list to return k elements.

In the biased approach the algorithm proceeds in a deterministic path across the stored data, so that if the database is not updated, two consecutive queries asking for the same predicate always return the same answer. As a consequence, some objects that could be part of the answer will never be retrieved, no matter how many times the query is posed. In the sampled tie list, the algorithm includes a random sampling technique to choose the elements of the L_t to assure that each query call will return correct but distinct answers whenever more than one exists.

The approach of choice depends on the application. To many applications, always returning the same answer is an undesirable effect. Common examples are those presenting large tie lists and few updates, such as systems storing health care data, specially those designed for teaching purposes. These databases have most of the attributes as categorical ones, leading to large number of ties, and as they store data from selected patients aiming illustration purposes, each one has few updates. Moreover, as the retrieved data is usually employed to feed the human interface modules of the application, the number of neighbors asked cannot allow too many samples, so the use of the tie list can be burdensome to the application and/or the human user. Therefore, presenting a variety of similar cases at different issues of the same query can be a valuable resource.

To other applications, having the same answer for the same query posed twice is a better option. In this case, the algorithm should prepare the answer in a deterministic way. This occurs for example if whenever an element is found to be inserted at the current tie list, it always replaces or always not replace those previously chosen to be given as part of the answer set. As both approaches are interesting to different applications, the algorithms presented in the next section embrace both of them. Therefore, the tie parameter of the $Nearest(s_q, k, tie)$ algorithm has its domain broadened to allow asking for the complete tie list ($tie = true$), a random sample ($tie = sample$) or a biased subset of the tie list ($tie = biased$).

Notice that neither approach guarantees a deterministic $kNNq$ answer, as updates in the database can change the results, even when the update does not change the $L_b + L_t$ result. Supporting a tie list does not increase significantly the computational cost of the $Nearest()$ algorithm since the number of disk accesses and number of distance calculations remain the same. The total time is only slightly larger in data sets with many ties. We show in Section 6 that this increase is indeed almost null.

5 Combining Similarity Operators: The New Algorithms

This section presents the $kAndRange()$ and the $kOrRange()$ algorithms. They follow the branch-and-bound approach and they are described here considering the data organized following a hierarchical MAM with every object stored at the leaf nodes, as the Slim-tree or the M-tree. However, the concept of the algorithms are independent of the particular MAM used and can also be applied on a non indexed data set. We present only the algorithms to search metric structures, as their implementation considering sequential scan can be developed straightforwardly. As the complex queries involve kNN predicates, the new algorithms were developed considering the processing of a tie list following the rules expressed as Equation 5.

For simplicity, we refer to the distance from the query center to an object as the radius of the object in the query. We assumed the Eu-

clidean metric to generate the figures of this section, so the radii are represented by circumferences. However, pay attention that, as any metric can be employed, the real shape of the covered areas depends on the metric used.

5.1 Conjunction of $kNNq$ and Rq predicates

The $kAndRange()$ algorithm performs conjunctive complex similarity query equivalent to a $R_q(s_q, r_q)$ **AND** a $kNNq(s_q, k, tie)$ where the query center is the same. It must recover every object that satisfies both basic similarity predicates, that is, the intersection of the intermediate results from both basic operators. Considering a data set S this can be defined as:

$$\sigma_{\left(R_q(s_q,r_q)\right)} S \cap \sigma_{\left(kNNq(s_q,k,tie)\right)} S \Leftrightarrow$$
$$\sigma_{\left(R_q(s_q,r_q)\ \wedge\ kNNq(s_q,k,tie)\right)} S \Leftrightarrow$$
$$\sigma_{\left(kAndRq(s_q,r_q,k,tie)\right)} S$$

where $kAndRq$ is the conjunctive predicate executed by the $kAndRange()$ algorithm.

The result of the conjunctive query satisfies the most restrictive condition between the two basic predicates involved, so the condition resulting in the smallest limiting radius contains the final answer: the radius of the k-th object of the nearest neighbor operator, which we call the nearest radius, or the query radius of the range operator, which we call the range radius. Figure 1 represents this idea, showing the three possible situations: a) range radius larger than nearest radius; b) range radius shorter than nearest radius; and c) range radius equal to nearest radius.

In Figure 1, s_q represents the query center, the continuous-line-border circle shows where the answer to the range predicate can be found, the dashed line border circle shows where the answer to the nearest neighbors predicate can be found, the gray circle is where the answer set of the complex query can be found, $numObj$ is the maximum number of objects recovered by a query, TL is the tie list and tie states if the tie list is required.

Figure 1.a represents the case when the answer is restricted by the nearest-neighbor condition. The answer set in Figure 1.b represents the case when the answer is restricted by the range predicate, and Figure 1.c shows the case when the range radius is equal to the radius of the k-th nearest neighbor, so the answer set contains every object that satisfies both predicates.

Notice that the number of objects retrieved by a conjunctive similarity query can change depending on whether the option tie is active or not, and on whether the answer set is bounded by the nearest-neighbor predicate or not, cases depicted in both Figure 1.a and 1.c. Figure 2 exemplify the case when the answer is restricted by the nearest-neighbor condition (case (a) in Figure 1) regarding a data set S containing seven elements, to answer the predicate $kAndRq(s_q, r_q, 2, tie)$. Notice that in this case the range condition is looser than the nearest condition, so if the tie list is not required, the number of objects returned $numObj = 2$ equals the required k, as shown in Figure 2.a. If the tie list is required, the number of objects returned can be larger than k to include the whole tie list, as shown in Figure 2.b.

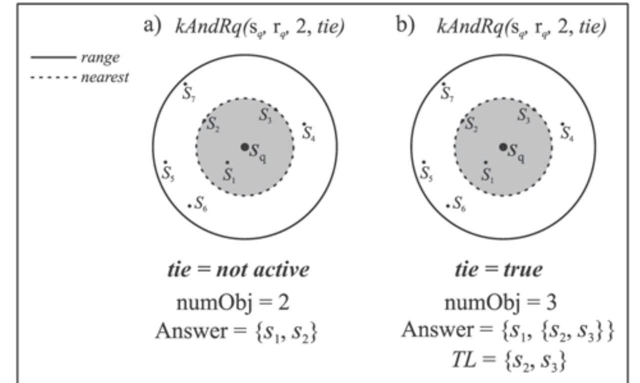

Figure 2: Tie list in conjunctive similarity queries where the range condition is looser than the nearest condition. a) with *tie=not active; b) with tie=true*

5.1.1 Algorithms to Manage the tie list

The answer of both the $kAndRange()$ and the $kOrRange()$ algorithms is a list $Answer$, which

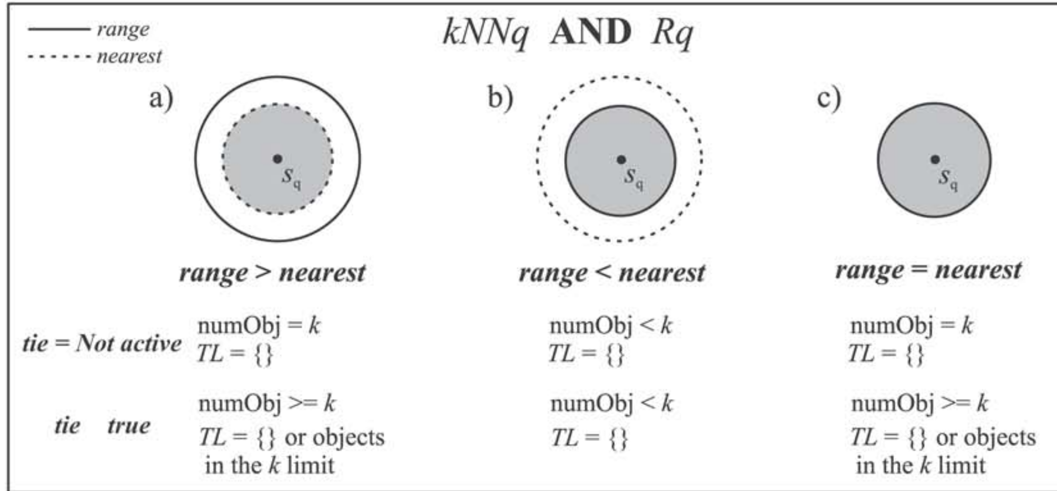

Figure 1: Graphical representation of the conjunction $kNN_q \wedge R_q$.

is kept sorted by the distances of each element s_i to the query center. This list is managed by the following methods: $Add(obj, distance)$ inserts a new element keeping the list sorted; $Length()$ returns the number of elements in the list; $DropLast(k, tie)$ removes the farthest element(s) in the list, maintaining the tie list; and $MaxDist()$ returns the largest distance from the query center to an element in the list. The tie list is kept in the end of this list, so the maximum number of elements stored can be larger than k. Therefore, before return, algorithms $kAndRange()$ and the $kOrRange()$ must check if a tie list is required and, if not, then the method $ChopAnswer(k)$ is called to choose a random or a biased subset of elements tied at the farthest distance to be returned.

Algorithms $Add()$, $Length()$ and $MaxDist()$ are straightforward to be implemented. Algorithm $DropLast()$ is shown as Algorithm 1. When called, it drops from the list every object farther than the object at position k from the query center. Algorithm $ChopAnswer()$ is shown as Algorithm 2. When tie asks for a biased subset of the tie list, it cuts every object after position k (steps 9 and 10). When tie asks for a sampled subset of the tie list, it randomly remove objects tied with the object at position k until only k objects remains (steps 2 to 7).

Algorithm 1 *Answer.DropLast(k, tie)*

1: $p := Answer.Length()$
2: **if** $Aux > k$ **then**
3:    **while** $Answer[k].dist < Answer[p].dist$ **do**
4:       $Answer.RemoveLast()$
5:       $p := p - 1$

Algorithm 2 *Answer.ChopAnswer(k)*

1: **if** tie =**sample then**
2:    $LargetDist := Answer[k].dist$
3:    **while** $Answer[k].dist = LargestDist \wedge k > 1$ **do**
4:       $k := k - 1$
5:    **while** $Answer.Length() > k$ **do**
6:       $p := Random[LargestDist, Answer.Length()]$
7:       $Answer.Delete(p)$
8: **else if** tie =**biased then**
9:    **while** $Answer.Length() > k$ **do**
10:       $Answer.RemoveLast()$

5.1.2 The $kAndRange()$ Algorithm

The $kAndRange(s_q, r_q, k, tie)$ algorithm, shown as Algorithm 3, executes the conjunction $Rq(s_q, r_q) \wedge kNNq(s_q, k, tie)$. It takes advantage of a global priority queue ($Queue$) to choose the paths that lead to best pruning.

The priority queue contains pointers to the active subtrees, i.e., subtrees where qualifying objects can be found. It has the following two methods: $Insert()$ to add a new active node; and $GetNode()$ to get the higher priority node. The priority is defined as the distance of the representative of the node s_{rep} to the query center, that is $d(s_{rep}, s_q)$.

The $kAndRange()$ is a recursive algorithm that receives the root $Node$ of the (sub-)tree to be traversed, and navigates down to the leaf nodes, applying the triangular inequality property to prune branches that do not store objects of the answer (see Algorithm 3). This algorithm returns the objects that are the nearest to the query center and that are also inside of the range radius.

The $kAndRange()$ algorithm starts reading the root node of the (sub-)tree to be traversed (line 1) and, using the priority queue $Queue$, navigates in deep-first mode down to the leaf nodes. It uses the triangular inequality property and the k and r_q limiting values to prune branches that cannot store objects of the answer. In a non leaf node (lines 14 to 19), this algorithm performs an ordered insertion of subtrees that could not be excluded by the triangle inequality.

Leaf nodes are handled at lines 3 to 13. If an object s_i in a leaf node cannot be pruned based on the distance between the node representative s_p and the query center s_q (Line 5), then the distance of the object s_i to the query center is calculated in Line 6. If it is inside the range radius r_q (line 7), s_i is put in the answer set. Line 6 checks if the size of the list holding the answer set is shorter than the required number k of nearest neighbors. If so the object s_i is added to the answer set (Line 8). Otherwise, Line 10 checks if the distance between this object and the query center is smaller or equal to the largest distance between the query center and the objects that are in the result list. When the condition in Line 10 is satisfied, the object s_i is added to the result list, which is kept sorted by the distances from each object to the query center (Line 11), and the $DropLast$ function is called (Line 12). This function uses the number k of nearest neighbors required and the variable tie to appropriately maintain the result list and the tie list, deleting objects in both if the newly inserted object reduces the current dynamic radius. Line 13 is an optimization step that reduces the query radius r_q if k elements nearer to the query center than r_q were already found.

Algorithm 3 The $kAndRange(s_q, r_q, k, tie)$ algorithm

1: $Queue.Insert(RootNode, 0)$
2: **while** ($Node := Queue.GetNode()) \neq Empty$ **do**
3:   **if** $Node$ is a leaf node **then**
4:     **for** each $s_i \in Node$ **do**
5:       **if** $|d(s_p, s_q) - d(s_i, s_p)| \leq r_q$ **then**
6:         Compute $d(s_i, s_q)$
7:         **if** $d(s_i, s_q) \leq r_q$ **then**
8:           **if** $Answer.Length() < k$ **then**
9:             $Answer.Add(s_i, d(s_i, s_q))$
10:          **else if** $d(s_i, s_q) \leq Answer.MaxDist()$ **then**
11:             $Answer.Add(s_i, d(s_i, s_q))$
12:             $Answer.DropLast(k, tie)$
13:             $r_q := Answer.MaxDist()$
14:   **else**
15:     **for** each $s_p \in Node$ **do**
16:       **if** $|d(s_p, s_q) - d(s_{rep}, s_p)| \leq r_q + r_{rep}$ **then**
17:         Compute $d(s_{rep}, s_q)$
18:         **if** $d(s_{rep}, s_q) \leq r_q + r_{rep}$ **then**
19:           $Queue.Insert(s_q, d(s_{rep}, s_q))$
20: **if** $tie \neq$ **true then**
21:   $Answer.ChopAnswer(k)$

5.2 Disjunction of $kNNq$ and Rq Predicates

The $kOrRange()$ algorithm performs disjunctive similarity query equivalent to a $R_q(s_q, r_q)$ **OR** a $kNNq(s_q, k, tie)$ where the query center is the same. It must recover every object that satisfies at least one of the complex query predicates, that is, the union of the intermediate results from both basic operators. Considering a data set S this can be defined as:

$$\sigma_{\left(R_q(s_q,r_q)\right)}S \cup \sigma_{\left(kNNq(s_q,k,tie)\right)}S \Leftrightarrow$$
$$\sigma_{\left(R_q(s_q,r_q)\ \vee\ kNNq(s_q,k,tie)\right)}S \Leftrightarrow$$
$$\sigma_{\left(kOrRq(s_q,r_q,k,tie)\right)}S$$

where $kOrRq$ is the disjunctive operator.

The result of the disjunctive query satisfies any of the two basic predicates involved, so the answer consists of the objects covered by the predicate with the largest limiting radius. Figure 3 represents this idea, using the same notation of Figure 1. The same three situations described in Section 5.1 occurs with disjunctive queries too.

The answer set shown in Figure 3.a is the range condition. The answer set in Figure 3.b includes every object that satisfies the nearest-neighbor condition, and Figure 3.c shows the case where both the range radius and the radius of the k-th nearest neighbors predicates are equal. The tie list is also considered and it can change the maximum number of recovered objects as in conjunctive queries, which in disjunctive predicates can occur in the case shown in Figure 3.b.

5.2.1 The $kOrRange()$ Algorithm

The $kOrRange(s_q, r_q, k, tie)$ algorithm, shown as Algorithm 4, executes the disjunction $Rq(s_q, r_q) \vee kNNq(s_q, k)$. It is similar to the $kAndRange()$ algorithm, and also uses a global priority queue ($Queue$) similar to the one used in the $kAndRange()$ algorithm to choose the paths that lead to the best pruning.

The $kOrRange()$ algorithm starts reading the root node of the (sub-)tree to be traversed (line 1) and, using the priority queue, navigates in deep-first mode down to the leaf nodes. As the range condition cannot define an upper-bound limit for the whole query, the dynamic radius is initially set to infinity in line 2. Whenever a non leaf node is read (lines 14 to 19), this algorithm inserts in $Queue$ the subtrees that could not be excluded by the triangle inequality.

Leaf nodes are handled in lines 3 to 16. If an object s_i in a leaf node cannot be pruned based on the distance between the node representative s_p and the query center s_q (Line 6), then the distance of the object s_i to the query center is calculated in Line 7. Whenever an object satisfies at least one of the operators, i.e., if object s_i is inside the dynamic radius d_k (line 8), it is added to the answer set (line 9). After the first k objects were already found and a new object is inserted, the exceeding elements must be deleted from the result (lines 10 to 12). However, this is done only if the d_k value is greater than r_q, otherwise the object is just inserted, to comply with the disjunction rule. Notice that, the dynamic radius d_k is progressively updated as more suitable elements are found during the search, but it never drops below r_q (lines 13 to 16).

Algorithm 4 $kOrRange(s_q, r_q, k, tie)$

```
 1: Queue.Insert(RootNode, 0)
 2: d_k := ∞
 3: while (Node := Queue.GetNode()) ≠ empty do
 4:   if Node is a leaf then
 5:     for each s_i ∈ Node do
 6:       if |d(s_p, s_q) − d(s_i, s_p)| ≤ d_k then
 7:         Compute d(s_i, s_q)
 8:         if d(s_i, s_q) ≤ d_k then
 9:           Answer.Add(s_i, d(s_i, s_q))
10:           if d_k > r_q then
11:             if Answer.Length() ≥ k then
12:               Answer.DropLast(k, tie)
13:               if Answer.MaxDist() ≤ r_q then
14:                 d_k := r_q
15:               else
16:                 d_k := Answer.MaxDist()
17:   else
18:     for each s_p ∈ Node do
19:       if |d(s_p, s_q) − d(s_rep, s_p)| ≤ r_rep + d_k then
20:         Compute d(s_rep, s_q)
21:         if d(s_rep, s_q) ≤ d_k + r_rep then
22:           Queue.Insert(s_q, d(s_rep, s_q))
23: if tie ≠ true then
24:   Answer.ChopAnswer(k)
```

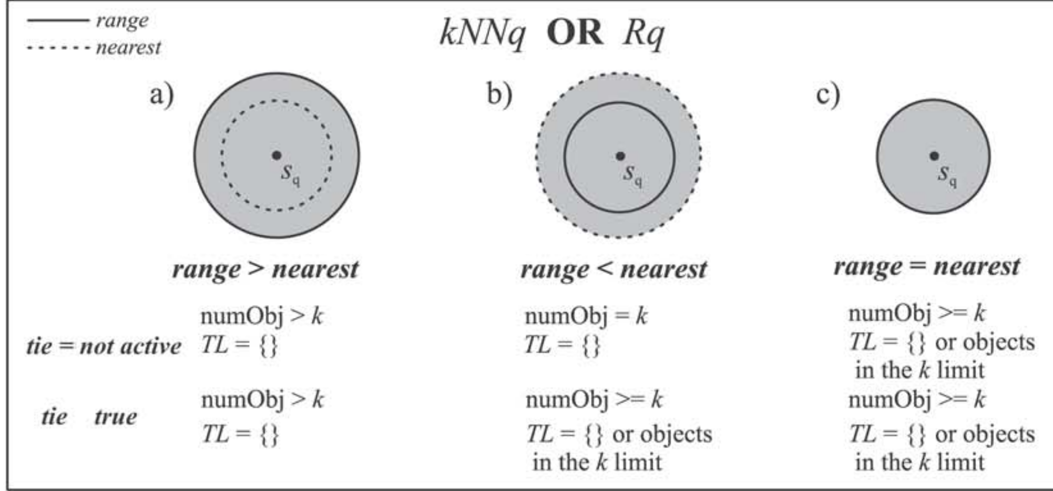

Figure 3: Graphical representation of the disjunction $kNN_q \wedge R_q$.

5.3 Considerations About the New Algorithms

Prunability. By comparing the two new algorithms with the basic algorithms discussed in Section 4, we can do the following considerations related to the query radius. The $kAndRange()$ algorithm always has a maximum radius defined in the query. Therefore, when the data set is indexed by a hierarchical MAM, it prunes subtrees with a high pruning ability (prunability), usually at a much higher rate than the one obtained by the basic nearest neighbor algorithm. This is due to the basic k-nearest neighgbor algorithm cannot use a limiting radius initially. In the experiments we verified that the $kAndRange()$ algorithm always has a prunability and a performance equivalent or better than those of the basic range query algorithm, which in turn usually have a much higher prunability then the basic $Nearest$ algorithm.

The $kOrRange()$ algorithm does not have a maximum radius defined in the query parameters, and presents a lower prunability than the $kAndRange()$ algorithm. However, as it also evaluates two predicates at once, it has a prunability higher than the prunability of the basic k-nearest neighbor algorithm.

Algebraic Rules. To define algebraic rules to guide optimization processes of complex queries is not the objective of this paper. However, the proposed algorithms were designed considering that algebraic rules could be applied. Therefore, we show here four algebraic rules that give an intuition of how these rules can be used by the query optimizer enabling the proposed algorithms to answer complex similarity queries. In fact, a complex similarity query involving two basic similarity predicates of the same type (that is, two Rq or two $kNNq$ predicates) with the same central object can be changed into one basic query of the same type. This can be performed by suitably choosing the respective radius or number of objects, observing the following rules for conjunctive or disjunctive similarity queries:

AND:

1. $\sigma_{(Rq(s_q,r_{q1}))}S \wedge \sigma_{(Rq(s_q,r_{q2}))}S \mid_{r_{q1} \leq r_{q2}} \Longleftrightarrow \sigma_{(Rq(s_q,r_{q1}))}S$
2. $\sigma_{(kNNq(s_q,k_1,tie))}S \wedge \sigma_{(kNNq(s_q,k_2,tie))}S \mid_{k_1 \leq k_2} \Longleftrightarrow \sigma_{(kNNq(s_q,k_1,tie))}S$

OR:

1. $\sigma_{(Rq(s_q,r_{q1}))}S \vee \sigma_{(Rq(s_q,r_{q2}))}S \mid_{r_{q1} \leq r_{q2}} \Longleftrightarrow \sigma_{(Rq(s_q,r_{q2}))}S$
2. $\sigma_{(kNNq(s_q,k_1,tie))}S \vee \sigma_{(kNNq(s_q,k_2,tie))}S \mid_{k_1 \leq k_2} \Longleftrightarrow \sigma_{(kNNq(s_q,k_2,tie))}S$

Using these rules, a query optimizer can change any expression involving multiple similarity predicates centered at the same query center into an expression that can be answered by the two proposed algorithms.

Multiple Centers. The composition considering more than one similarity predicate of the same type (Rq or $kNNq$) with distinct query objects can also be changed into only one query of the same type. This can be performed by a suitable choice from respective radius or object numbers, applying the algorithms proposed in this paper, and filtering their results. For example, two range queries can be executed by choosing one of them as the complex query center, and setting the query radius as the summation of their basic radii extended by the distance between the centers of the complex similarity query. After executing the changed query, the results are compared to each one of the original centers, thus filtering the final answer. The calculation of the corresponding query radius or object numbers to be used in the algorithms proposed in this paper and the filtering of their results can be executed using algebraic rules. This allows using the proposed algorithms to answer any complex similarity query.

6 Experiments

This section presents experimental measurements on the proposed algorithms, comparing them with the correspondent measurements obtained by compositions of the basic algorithms. Every algorithm was implemented in two versions: through a sequential scanning over the data set (SeqScan) and using the Slim-tree metric access method. The algorithms were implemented in C++, and the experiments were run in an Intel Pentium-4 1.6GHz machine, with 512MB of RAM memory and a 40GB disk spinning at 7200RPM, under the Microsoft Windows 2000 operating system. The following subsection presents the settings that we have used in experiments and Subsection 6.2 shows the measurements obtained.

6.1 Experimental Setup

To evaluate the performance and efficiency of the proposed algorithms, we have used a variety of data sets, both synthetic and from the real world, although in this paper we present only the results obtained from the following four data sets.

- *Synthetic* - a synthetic set of points uniformly distributed in 6-dimensions;
- *LBeach* - a set of geographical points in a 2-dimensional space describing the coordinates of the road intersections in Long Beach City, CA, from the TIGER system of the U.S. Bureau of Census;
- *CorelHisto* - a set of attributes describing colors in images in 32 dimensions, from the UCI repository (kdd.ics.uci.edu);
- *Words* - a set of words extracted from a Portuguese language dictionary [20].

For the *Words* data set we used the Levenshtein metric. The other are dimensional data sets, so we used the Euclidean metric (L_2). The object size changes in each data set, so the maximum node capacity of the Slim-tree changes too, to test trees using the same node size of 4kBytes. The properties of the four data sets and the maximum node capacity are summarized in Table 3.

Table 3: Data sets used in the experiments.

Name	# of objects	Metric	Node Capacity
Synthetic	50,000	L_2	68
LBeach	36,298	L_2	72
CorelHisto	68,040	L_2	60
Words	21,223	L_{Edit}	50

The $kAndRange()$ and $kOrRange()$ algorithms were compared with the equivalent composition of the basic algorithms producing the same answer set. A union/intersection of the basic algorithms corresponds to execute the $Nearest()$ algorithm, the $Range()$ algorithm

and then the set union or the set intersection operator. The set operators are performed in memory.

Every measurement represents the average of 500 queries regarding the number of distance calculations, the number of disk accesses and the total time in milliseconds. Each set of 500 queries has its query center object chosen in the following way: 250 were sampled from the respective data set but were kept in the data set; the other 250 objects were sampled from the respective data set and removed from it. Therefore, the queries cover both the biased queries regarding the distribution of the data set elements and the randomly distributed queries scattered in the data domains, both of them occurring in real applications. Each measurement considers a range radius r_q and a fixed number k of nearest neighbors, averaging the results over the set of 500 query centers. In each plot, the abscissa represents the number of objects retrieved, expressed as a percentile of the data set.

In each experiment, the number of neighbors and the range radius varies as follows. The radius for the *Synthetic*, *LBeach* and *CorelHisto* data sets varies from 0.01% up to 10% of the data set diameter. For the *Words* data set the radius varies from 1 up to 10 editions. The values of k for the *Synthetic*, *LBeach* and *CorelHisto* is respectively 0.01%, 0.02%, 0.05% of the data set, and is 5 words for the *Words* data set.

6.2 Performance Evaluation

This section presents the results obtained from the $kAndRange()$, $kOrRange()$ and the basic algorithms both searching a Slim-tree and through the SeqScan.

Figure 4 compares the proposed and the basic algorithms to answer 500 queries in the four data sets. The plots are presented in log-log scale for the *Synthetic*, *LBeach*, *CorelHisto* data sets, and in linear-log scale for the *Words* data set. Every experiment asks for the tie list. The figure shows the plots of the average number of disk accesses (Figures 4.A, D, G and J), the average number of distance calculations (Figures 4.B, E, H and K), and the total time (Figures 4.C, F, I and L).

Considering the SeqScan, the union/intersection of the basic algorithms (plots F and H in the graphs) and $kOrRange()$/$kAndRange()$ algorithms (plots E and G) present a constant number of disk accesses and distance calculations considering query range and k. However, the traditional approach requires twice as many distance calculations and number of disk accesses as our proposed algorithms. Searching the Slim-tree using either the union or intersection of basic algorithms (plots B and D) has the same number of disk accesses and of distance calculations. However, either the $kAndRange()$ or $kOrRange()$ algorithms (plots A and C) requires up to 30% less disk acesses and distance calculations. Moreover, the $kAndRange()$ algorithm grow linearly up to the maximum variation of k and then assumes a sub-linear behavior, and tends to stabilize (plot A). This is easily seen in the *CorelHisto* data set, as shown in Figures 4.G and 4.H.

The new algorithms have a different behavior when using the Slim-tree in the *Words* data set (Figures 4.J and 4.L), as both algorithms have closer behavior and their numbers of disk accesses and distance calculations quickly become constant. This happens because the results of the L_{Edit}distance function give discrete values and produces many ties. Figure 4.J shows that the Slim-tree requires a number of disk accesses larger than the SeqScan. This is because the Slim-tree requires more disk space to store the structure itself. However, as the number of distance calculations drops as shown in 4.L, the total time of every algorithm is smaller in the Slim-tree than in its SeqScan counterpart (4.M).

The behavior of each algorithm regarding the numbers of disk accesses and distance calculations influences its total time, as is shown in 4.C, 4.F, 4.I and 4.M. When the radius grows, the total time grows too, as is evidenced in the algo-

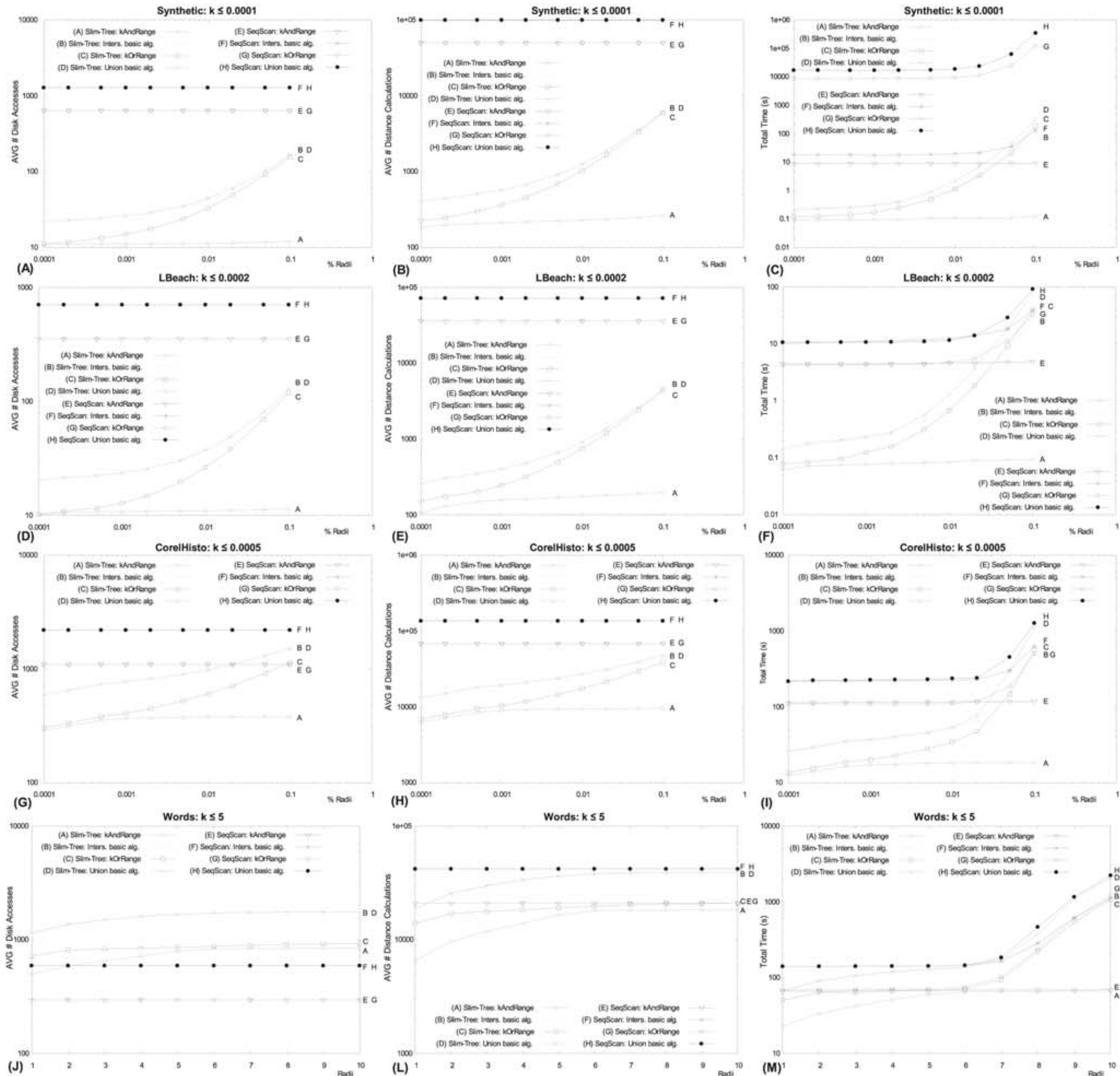

Figure 4: Comparing the performance to answer 500 queries using the proposed and the basic algorithms using the Slim-tree and the SeqScan, on the *Synthetic*, *LBeach*, *CorelHisto* and *Words* data sets. (A, D, G, J) Average number of disk accesses per query. (B, E, H, K) Average number of distance calculations per query. (C, F, I, L) Total time for 500 queries in seconds.

rithms searching a Slim-tree. Besides, it must be noticed that the union/intersection operations just affect the time measurements of the SeqScan accesses (plots E, F, G and H). When searching a Slim-tree, the $kAndRange()$ algorithm is at least 2 times faster for small radii (plots A, B, C and D), increasing to be up to 14 times faster for larger radii than the intersection of basic algorithms counterpart, as shown in Figure 4.C for the *Synthetic* data set. The $kOrRange()$ Algo-

rithm is about two times faster than the union of the basic algorithms for small radii, increasing to be up to 3 times faster for large values of radii, as shown in Figure 4.F for the *LBeach* data set.

Regarding the Slim-tree, the new algorithms provide higher improvements in every measured aspect. The $kAndRange()$ algorithm requires at most half the numbers of distance calculations, of disk accesses and total time to process queries with small radii in every data set tested, as comparing with the intersection of the basic algorithms (plots A and B). For higher values of radii, e.g., when retrieving 10% of the data set, the reduction is even larger: it reduces to $1/12$ of the number of disk accesses in the *LBeach* data set (Figure 4.D), to $1/23$ of the number of distance calculations in *Synthetic* data set (Figure 4.B), and to less than one hundredth of the total time in the *LBeach* and *Synthetic* data sets (Figures 4.C and 4.F). The $kOrRange()$ algorithm also requires equivalent reduction compared to the union of the basic algorithms for small radii in almost all data sets (plots C and D). The exception happens to the *Words* data set, where for small radii the reduction is larger, as the $kOrRange()$ algorithm reduces to almost 50% every measured aspect (Figures 4.J, 4.L and 4.M). For small values of radii (less than 0,2% of the data set, the gain of the $kOrRange()$ algorithm decreases to only 10% in the number of disk accesses and distance calculations in *LBeach* and *Random* data sets (Figures 4.A, 4.B, 4.D and 4.E). However, the *Words* data set is again an exception, once this new algorithm achieved improvements of at least twice in every measured aspect as compared with the union of the basic algorithms.

It is interesting to note that the proposed algorithms provide the most remarkable improvements in the total time for the queries most frequently used in real systems, i.e., those queries with small range radius and few neighbors. Moreover, the experiments show that every measurement performed using the $kAndRange()$ or $kOrRange()$ algorithms presented better performance than the correspondent intersection or union of the basic algorithms, either searching a Slim-tree or using a SeqScan.

In another experiment using the *Synthetic* data set, we generated the data set in 10 steps, adding 10,000 elements at each step, and for each database size we measured the total time to calculate 500 queries using $k = 0,1\%$ of the number of elements in the database and $r_1 = 0,1\%$ of the data set diameter. The result, shown in Figure 5, shows that the proposed algorithms present linear behavior when varying the data set size, so they are scalable regarding the data set size. It also shows that as the data set size increases, the more important is the use of a MAM.

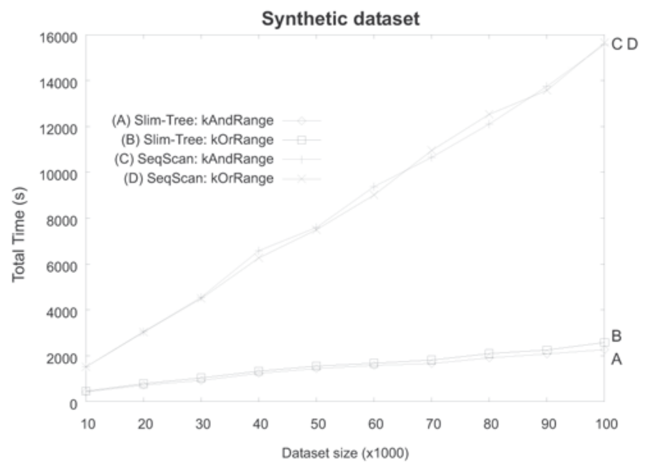

Figure 5: Measurement of wall-clock time to execute 500 queries using the $kAndRange()$ and the $kOrRange()$ algorithms using both SeqScan and Slim-tree with varying data set size, showing the scalability of the algorithms.

6.2.1 Measurements involving the tie list

In this subsection we show the impact of retrieving tie lists on the performance of complex algorithms to answer similarity queries. The number of disk accesses and the number of distance calculations remains the same whether a tie list is required or not, in both $kAndRange()$ and $kOrRange()$ algorithm. This is due to the fact

that both algorithms prune subtrees that are farther than the region covering the algorithm's limiting radius, but does not exclude nodes that are at nearer or at equal distance as the limiting radius. Thus, when an object in a leaf node qualifies, it is added in the answer set and only then the $Answer.DropLast()$ method can check the tie variable to determine if the new object will be maintained in the answer set or not. Therefore, the numbers of disk accesses and distance calculations are not affected, which was confirmed in the experimental measurements.

However, the total time can change when a tie list is required. This happens because managing the answer set is slightly more complex when the tie list is required. The experiments show that total time increases proportionally to the number of ties, but very slowly. Although we evaluated every data set presented in this paper, only the *Word* data set presented measurable differences.

Figure 6 shows the total time in conjunctive and disjunctive queries measured in the *Words* data set, comparing when asking for a biased tie list, for a sampled tie list or for no tie list. Both $kAndRange()$ and $kOrRange()$ algorithms were tested searching a Slim-tree, because being it faster, processing the tie list have a larger impact in the total time. Notice that the increase in total time is so slightly that it is barely visible in Figure 6. Numerically, the $kAndRange()$ algorithm takes 59.92s to calculate 500 queries with $k = 3$ and $r_q = 9$ without a tie list. The total time increases 0.063s when asking for a biased tie list, and 0.344s when asking for a sampled tie list. The $kOrRange()$ algorithm takes 2055.95s to calculate 500 queries with $k = 3$ and $r_q = 9$ without a tie list, increasing 0.078s when asking for a biased tie list, and 6.640s when asking for a sampled tie list.

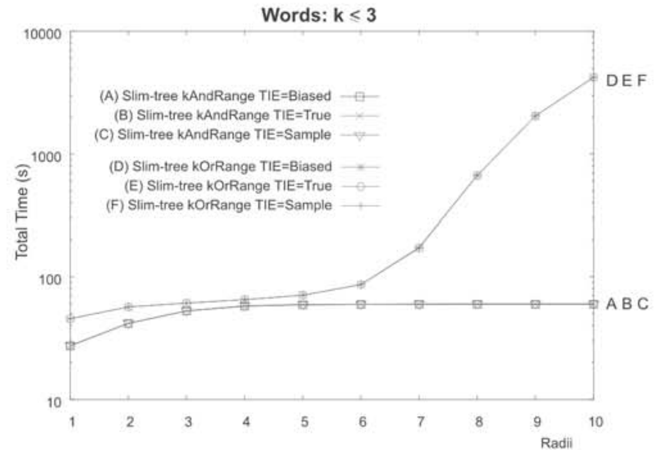

Figure 6: Comparing total time to answer 500 queries using $kAndRange()$ and $kOrRange()$ algorithms over the Slim-tree, with and without the tie list.

7 Conclusions

This paper presented two new algorithms, called *kAndRange()* and *kOrRange()*, that were developed to support the composition of similarity operators using conjunction and disjunction between range and k-nearest neighbor conditions in complex similarity queries centered at the same query object. These algorithms were created aiming to support a tie list in the result. A tie list enables to control two problems existing in algorithms that involves the *kNNq* operator: the non repeatability answering similarity queries and the hiding of results that can be relevant in queries, as discussed in Section 4.1.

The experiments presented confirm that the proposed algorithms are scalable and more efficient than the correspondent composition of the basics algorithms to answer complex similarity queries. In addition, the measurements related to the tie list show that supporting it does not require additional cost in query processing. The experimental measurements demonstrate a consistent improvement in answering complex similarity queries and their efficiency and scalability. The experiments also show that the new algorithms reduce the total time and numbers of disk accesses and distance calculations to at most half, improving the most frequent queries posed in real systems. Moreover, the experiments showed that

the new algorithms reduced up to 12 times the number of disk accesses, more than 20 times the number of distance calculations and can be more than a hundred times faster than the correspondent composition of the basic algorithms.

Notice that to answer those queries without the new algorithms it is necessary to run the basic algorithms individually, composing the intermediate results through intersections or unions to produce the final result. Therefore, the main contribution of this paper is enabling RDBMS to perform complex similarity queries in a practical way, through the inclusion of the similarity operators as an extension of SQL. In addition, this paper makes it possible to develop desirable characteristics in similarity queries as future works, such as the support for a query optimizer to handle similarity queries through a set of algebraic rules covering similarity predicates, changing of any expression involving multiple similarity predicates centered at the same query center into an expression that can be answered by the two proposed algorithms, or developing the support for the composition of similarity queries with distinct centers, as briefly suggested in Section 5.3.

Acknowledgements

This work has been supported by FAPESP (São Paulo State Research Foundation) under grants 01/02426-8, 01/11987-3, 02/07318-1 and by **CNPq** (Brazilian National Council for Supporting Research) under grants 52.1685/98-6, 860.068/00-7, 50.0780/2003-0 and 35.0852/94-4.

References

[1] 13249-3:2001, I. (2001). Information technology – Database Languages – SQL Multimedia and Application Packages – part 3: Spatial – ISO/IEC.

[2] Arantes, A. S., Vieira, M. R., Traina Jr., C., and Traina, A. J. M. (2003). Operadores de seleção por similaridade em sistemas de gerenciamento de bases de dados relacionais. In *Proc. of the XVIII SBBD*, pages 341–355, Manaus, Brasil.

[3] Araujo, M. R. B., Traina, A. J. M., and Traina, Caetano, J. (2002). Extending SQL to support image content-based retrieval. In *Proc. of IASTED ISDB*, page 6, Tokyo, Japan.

[4] Benetis, R., Jensen, C. S., Karciauskas, G., and Saltenis, S. (2002). Nearest neighbor and reverse nearest neighbor queries for moving objects. In Nascimento, M. A., Özsu, M. T., and Zaiane, O. R., editors, *Proc. of IDEAS'02*, pages 44–53, Edmonton, Canada. IEEE Computer Society.

[5] Berchtold, S., Ertl, B., Keim, D. A., Kriegel, H.-P., and Seidl, T. (1998). Fast nearest neighbor search in high-dimensional space. In *Proc. of the 14th ICDE*, pages 209–218, Orlando, USA. IEEE Computer Society.

[6] Böhm, K., Mlivoncic, M., Schek, H.-J., and Weber, R. (2001). Fast evaluation techniques for complex similarity queries. In Apers, P. M. G., Atzeni, P., Ceri, S., Paraboschi, S., Ramamohanarao, K., and Snodgrass, R. T., editors, *Proc. of the 27th VLDB*, pages 211–220, Roma, Italy. Morgan Kaufmann.

[7] Chaudhuri, S. and Gravano, L. (1996). Optimizing queries over multimedia repositories. In *Proc. of SIGMOD*, pages 91–102, Quebec, Canada.

[8] Cheung, K. L. and Fu, A. W.-C. (1998). Enhanced nearest neighbour search on the R-tree. *ACM SIGMOD Records*, 27(3):16–21.

[9] Chávez, E., Navarro, G., Baeza-Yates, R., and Marroquín, J. L. (2001). Searching in metric spaces. *ACM Computing Surveys (CSUR)*, 33(3):273–321.

[10] Ciaccia, P., Patella, M., and Zezula, P. (1997). M-tree: An efficient access method for similarity search in metric spaces. In Jarke, M., editor, *Proc. of the 23th VLD)*, pages 426–435, Athens, Greece. Morgan Kaufmann Publishers.

[11] Ciaccia, P., Patella, M., and Zezula, P. (1998). Processing complex similarity queries with distance-based access methods. In *Proc. of EDBT*, volume 1377, pages 9–23, Valencia, Spain.

[12] Fagin, R. (1996). Combining fuzzy information from multiple systems. In *Proc. of ACM SIGMOD-PODS*, pages 216–226, Montreal, Canada.

[13] Faloutsos, C. (1997). Indexing of multimedia data. In *Multimedia Databases in Perspective*, pages 219–245. Springer Verlag.

[14] Guttman, A. (1984). R-tree: A dynamic index structure for spatial searching. In Yormack, B., editor, *Proc. of the 1984 ACM-SIGMOD*, pages 47–57, Boston, USA. ACM Press.

[15] Hibino, S. and Rundensteiner, E. A. (1998). Processing incremental multidimensional range queries in a direct manipulation visual query. In *Proc. of the 14th IEEE-ICDE*, pages 458–465, Orlando, USA. IEEE Computer Society.

[16] Hjaltason, G. R. and Samet, H. (1999). Distance browsing in spatial databases. *ACM Transactions on Database Systems (TODS)*, 24(2):265–318.

[17] Katayama, N. and Satoh, S. (1997). The SR-tree: An index structure for high-dimensional nearest neighbor queries. In Peckham, J., editor, *Proc. of the 1997 ACM-SIGMOD*, pages 369–380, Tucson, USA. ACM Press.

[18] Korn, F., Sidiropoulos, N., Faloutsos, C., Siegel, E., and Protopapas, Z. (1996). Fast nearest neighbor search in medical image databases. In Vijayaraman, T. M., Buchmann, A. P., Mohan, C., and Sarda, N. L., editors, *Proc. of the 22th VLDB*, pages 215–226, Mumbai (Bombay), India. Morgan Kaufmann.

[19] Melton, J. and Eisenberg, A. (2001). SQL multimedia and application packages (SQL/MM). *ACM SIGMOD Records*, 30(4):97–102.

[20] Nunes, M. G. V., Vieira, F. M. C., Zavaglia, C., Sossolote, C. R. C., and Hernandez, J. (1996). A construção de um léxico da Língua Portuguesa do Brasil para suporte à correção automática de textos. Relatórios técnicos do ICMC, USP - São Carlos - SP.

[21] Park, D.-J. and Kim, H.-J. (2003). An enhanced technique for k-nearest neighbor queries with non-spatial selection predicates. *Multimedia Tools and Applic.*, 19(1):79–103.

[22] Roussopoulos, N., Kelley, S., and Vincent, F. (1995). Nearest neighbor queries. In Carey, M. J. and Schneider, D. A., editors, *Proc. of the 1995 ACM SIGMOD*, pages 71–79, San Jose, USA. ACM Press.

[23] Traina, Caetano, J., Traina, A. J. M., Faloutsos, C., and Seeger, B. (2002). Fast indexing and visualization of metric datasets using Slim-trees. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(2):244–260.

[24] Traina, Caetano, J., Traina, A. J. M., Seeger, B., and Faloutsos, C. (2000). Slim-trees: High performance metric trees minimizing overlap between nodes. In Zaniolo, C., Lockemann, P. C., Scholl, M. H., and Grust, T., editors, *Proc. of EDBT'2000*, volume 1777 of *LNCS*, pages 51–65, Konstanz, Germany. Springer.

[25] White, D. A. and Jain, R. (1996). Similarity indexing with the SS-tree. In Su, S. Y. W., editor, *Proc. of the 12th ICDE*, pages 516–523, New Orleans, USA. IEEE Press.