Efficiently Supporting Multiple Similarity Queries for Mining in Metric Databases

Bernhard Braunmüller, Martin Ester, Hans-Peter Kriegel, Jörg Sander
Institute for Computer Science, University of Munich
Oettingenstr. 67, D-80538 München, Germany
email: {braunmue | ester | kriegel | sander}@dbs.informatik.uni-muenchen.de

Abstract

Metric databases are databases where a metric distance function is defined for pairs of database objects. In such databases, similarity queries in the form of range queries or *k*-nearest neighbor queries are the most important queries. In traditional query processing, single queries are issued independently by different users. In many data mining applications, however, the database is typically explored by iteratively asking similarity queries for answers of previous similarity queries. In this paper, we introduce a generic scheme for such data mining algorithms and we develop a method to transform such algorithms in a way that they can use multiple similarity queries, i.e. sets of queries issued simultaneously. We investigate two orthogonal approaches, reducing I/O cost as well as CPU cost, to speed-up the processing of multiple similarity queries. The proposed techniques apply to any type of similarity query and to an implementation based on an index or using a sequential scan. Parallelization yields an additional impressive speed-up. An extensive performance evaluation confirms the efficiency of our approach and we conclude that multiple similarity queries should be provided as a basic DBMS operation in order to support many data mining applications in metric databases.

1 Introduction

Metric databases are databases where a metric distance function is defined for pairs of database objects. A prominent special case are databases of objects from a vector space, that is objects with numeric attributes. For example, multimedia objects [Fal+ 94] are typically represented by a large number of numeric features such as shape descriptors or color histograms. In many scientific applications, e.g. in astronomy [Hog 97], automatic facilities measure a large number of numeric values for each database object such as the amplitude emitted in some frequency band. On the other hand, in a database monitoring WWW accesses, the objects may model URLs and these objects are not from a vector space but a metric distance function can be supplied.

Similarity between database objects is expressed by the distance function such that a low distance corresponds to a high degree of similarity whereas two objects with a large distance are considered to be rather dissimilar. *Similarity queries* [WSB 98], e.g. range queries or *k*-nearest neighbor queries, are the most important queries in metric databases. Such queries play a major role in applications such as multimedia systems, decision support systems and data mining.

A lot of research on analyzing large databases - manually and automatically - has been conducted. *Data exploration* is the process of manually exploring a database [KLTW 96]. A user starts at a given database object and from there he or she interactively navigates through the database, for example by iteratively retrieving all similar objects. That is, the answers of previous queries may be used as query objects for new similarity queries. *Knowledge discovery in databases* (*KDD*) has been defined as the non-trivial process of discovering valid, novel, potentially useful, and ultimately understandable patterns from data [FPS 96]. The core step of the KDD process

is the step of *data mining*, i.e. the application of appropriate algorithms that automatically produce a particular enumeration of patterns over the data. For example, a density-based clustering algorithm such as DBSCAN [EKSX 96] starts from some object and repeatedly retrieves the neighborhood of objects which have been retrieved by previous queries as long as the density in this neighborhood is large enough.

In traditional query processing, single queries are issued independently by different users. In manual data exploration as well as in automatic data mining, however, many similarity queries must be answered in a single application. We define *multiple queries* as sets of queries issued simultaneously. Clearly, multiple queries provide much more potential for query optimization than single queries. In this paper, we investigate two orthogonal approaches to speed-up the processing of multiple similarity queries in metric databases: reduce I/O cost (that is, the number of disk accesses) and reduce CPU cost (that is, the number of distance calculations). Furthermore, we explore the potential of parallelization. The proposed techniques can be combined in order to obtain a maximum performance of processing multiple similarity queries.

The rest of this paper is organized as follows. In section 2, we briefly review the standard methods of processing single similarity queries and we introduce some basic notions and algorithms. Section 3 introduces an algorithmic scheme typical for many data mining applications and discusses several instances of this scheme. Furthermore, we develop a method to transform such algorithms so that they issue sets of multiple queries. The concepts of multiple similarity queries are introduced in more detail in section 4. Section 5 presents several techniques for efficiently supporting multiple similarity queries. An extensive performance evaluation of the proposed techniques on two real databases is presented in section 6. Section 7 summarizes the contributions of this paper and outlines some issues for future research.

2 Processing Single Similarity Queries

Let the database objects be drawn from a set of *Objects* and let *dist* be a metric distance function for pairs of objects, i.e. *dist*: $Objects \times Objects \rightarrow \Re^+$, and *dist* satisfies the following conditions. $\forall\ O_1, O_2, O_3 \in Objects$:

1) $dist(O_1, O_2) = 0 \Leftrightarrow O_1 = O_2$. (identity)

2) $dist(O_1, O_2) = dist(O_2, O_1)$. (symmetry)

3) $dist(O_1,O_3) \le dist(O_1,O_2) + dist(O_2,O_3)$. ($\Delta$-inequality)

Often, the Euclidean distance or a weighted Euclidean distance is used as the distance function but, depending on the application, other distance functions may be more appropriate. For instance, quadratic form distance functions were successfully applied for an image database using color histograms as features [SK 97].

Definition 1: (*similarity query*)
Let $DB \subseteq Objects$ be a database and let $Q \in Objects$ be a query object. Let *T* denote the type of the similarity query and let sim_T: $Objects \times Objects \rightarrow Boolean$ be a predicate defining the similarity of pairs of objects wrt. to the type *T*. A *similarity query*, denoted as *DB.similarity_query(Q,T)*, returns the following database objects: $DB.similarity_query(Q, T) = \{O \in \mathrm{DB} \mid sim_T(O, Q)\}$.

The specification of the query type *T* consists of three components:

- *T.range*: a real number specifying a maximum distance between *Q* and an answer.
- *T.cardinality*: an integer number defining the maximum cardinality of the set of answers.

- *T.kind*: a string indicating how to combine the range condition and the cardinality condition.

The well-known types of similarity queries are obtained by different specializations of the query type.

Definition 2: (*range query*)

A *range query* with respect to a database $DB \subseteq Objects$ and a query object $Q \in Objects$ is a similarity query with *T.range* = ε, *T.cardinality* = +∞ and *T.kind* = "range" which returns the following subset of database objects:
$DB.similarity_query(Q, T) = \{O \in \mathrm{DB} \mid dist(O, Q) \leq \varepsilon\}$.

Definition 3: (*k-nearest neighbor query*)

A *k-nearest neighbor query* with respect to a database $DB \subseteq Objects$ and a query object $Q \in Objects$ is a similarity query with *T.range* = +∞, *T.cardinality* = *k* and *T.kind* = "k-nearest neighbor" which returns a set $NN_Q(k) \subseteq \mathrm{DB}$ that contains *k* objects from the database and for which the following condition holds:
$\forall P \in NN_Q(k), \forall O \in \mathrm{DB} - NN_Q(k): dist(P, Q) \leq dist(O, Q)$.

Other types of similarity queries have been proposed in the literature. For instance, we may be interested in the *k*-nearest neighbors but only in those within a specified range.

In order to speed-up similarity query processing, many spatial index structures (for good surveys see [Sam 89], [GG 98]) have been developed which are applicable for the important special case where the database objects are from a vector space. For instance, the *R-tree* [Gut 84] generalizes the one-dimensional B-tree to *d*-dimensional data spaces, that is an R-tree manages *d*-dimensional hyper-rectangles instead of one-dimensional numeric keys.

The *R-tree* and its variants such as the *R*-tree* [BKSS 90] are efficient only for relatively small numbers of dimensions *d*. Recently, index structures have been designed which are also efficient for some larger values of *d*. For instance, the *X-tree* [BKK 96] is similar to an R*-tree but introduces the concept of *supernodes*, i.e. nodes of variable size in the directory of the tree. Directory nodes are "merged" into one supernode, i.e. directory nodes are *not* split, if there is a high probability that all parts of the node have to be searched anyway for most queries.

In [Kei 97] and [WSB 98], it is shown that under the assumption of uniformly distributed data, above a certain dimensionality no index structure can process a nearest neighbor query efficiently. Thus, it is suggested to use the sequential scan which obtains at least the benefits of sequential rather than random disk I/O. In the *VA-file* [WSB 98], clever bit encodings of the data are used to speed-up the scan.

The above index structures are only applicable for vector spaces. The more general case of metric databases, however, is also important in applications such as WWW access log databases. Then, the database objects may be sessions grouping all log entries with identical IP address and user id within a given maximum time gap [MJHS 96]. General metric databases require other types of index structures, the so called metric trees. In these metric trees the *triangle inequality* is used to prune the search tree while processing a similarity query. Most of these structures are static in the sense that they do not allow dynamic insertions and deletions of objects. A recent paper [CPZ 97] has introduced a dynamic metric index structure, the *M-tree*, which is a balanced tree that can be managed on secondary memory. The leaf nodes of an M-tree store all the database objects. Directory nodes store so-called *routing objects* and associated covering radii to guide the search operations.

Query processing using a sequential scan is straightforward: all objects must be visited to answer a query. When using an index it may be possible to exclude large proportions of the data from a search. To answer a range query using a tree-based index, the set of approximations (e.g. hyper-rectangles in case of an R-tree) intersecting the query region is determined recursively starting from the root. In a directory node, the entries intersecting the query region are determined and then their referenced child nodes are searched until the data pages are reached. The procedure is more sophisticated for a *k*-nearest neighbor query because we do not know the *k*-nearest neighbor distance in advance. The algorithm proposed by [HS 95] has been proven in [BBKK 97] to minimize the number of pages read from disk. This algorithm processes the data pages in ascending order of distance from the query point and does not load those pages with an approximation further away than the *k*-nearest neighbor found so far.

To conclude this section, we present an algorithm for processing single similarity queries. This algorithm is applicable for any type of similarity query and it can be implemented either by using an index structure or by performing a sequential scan. Figure 1 presents the algorithm as a method of the class DB (database). It takes two arguments, a query object Q and a query type T, and returns a list of objects answering the similarity query.

```
DB::similarity_query(object Q; type T)
    Answers := initialize_answer_list();
    determine_relevant_data_pages(Q, T);
    QueryDist :=T.Range;
    while Self.unprocessed_pages() do
        NextPage := read_next_page_from_disk();
        for each object O in NextPage do
            Distance := dist(O,Q);
            if Distance ≤ QueryDist then
                Answers.insert(O); // in ascending order of dist(A,Q)
                if Answers.cardinality() > T.Cardinality() then
                    Answers.remove_last_element();
                    QueryDist := adapt_query_dist(Distance,QueryDist,T);
                    Self.prune_pages(QueryDist);
    return Answers;
```

Figure 1: Algorithm *similarity_query*

We discuss the most important details of this algorithm. The method DB::determine_relevant_data_pages(Q,T) is based on the algorithm presented in [BBKK 97] and it constructs a sequence of the physical addresses of all data pages which may contain answers of the similarity query specified by Q and T. Note that the resulting sequence is managed as a private attribute of the class DB which is read by the method DB.unprocessed_pages() and which is updated by DB.read_next_page_from_disk(). If the implementation makes use of an index structure, then a subset of all data pages of DB may be recognized as irrelevant and thus is not returned. Otherwise, if a sequential scan is performed, all data pages of DB are relevant. In both cases, the relevant data pages are ordered according to their physical address such that the number of disk seeks is minimized. DB.adapt_query_dist(Distance,QueryDist,T) changes the QueryDist only in the case of a *k*-nearest neighbor query but not in the case of a range query. DB.prune_pages(QueryDist) removes all elements page from the internal DB attribute of relevant data pages satisfying dist(page,Q) > QueryDist. Clearly, this method performs no operation if QueryDist has not been adapted.

3 Data Mining Using Multiple Similarity Queries

In many data mining applications the database is explored by iteratively considering the neighborhood of some start objects. In this section, we introduce a generic scheme for such data mining algorithms and discuss some typical instances of the scheme. Furthermore, we develop a method to transform such algorithms in a way that they can use multiple similarity queries instead of single similarity queries.

3.1 Iterative Neighborhood Exploration

Many data mining algorithms start from a set of specified database objects and iteratively consider the neighborhood of the visited objects. The *neighborhood* of a given object is defined as the set of similar database objects with respect to a similarity query. We introduce a generic scheme for such algorithms which we call *ExploreNeighborhoods*. Figure 2 depicts the algorithmic scheme in pseudo code notation where DB denotes a database, StartObjects denotes a subset of DB and SimType specifies the type of similarity query. The dots in the argument list of some functions indicate additional arguments which may be necessary for different instances of this algorithmic scheme.

```
ExploreNeighborhoods(DB, StartObjects, SimType, ...)
    ControlList := StartObjects;
    while ( condition_check(ControlList, ...) = TRUE ) do
        Object := ControlList.choose(...);
        proc_1(Object, ...);
        Answers := DB.similarity_query(Object, SimType);
        proc_2(Answers, ...);
        ControlList := ( ControlList » filter(Answers, ...) ) - {Object};
    end while;
```

Figure 2: Algorithmic scheme *ExploreNeighborhoods*

Starting from the objects in the set StartObjects, the algorithm repeatedly retrieves the neighborhood of objects taken from the ContolList as long as the function condition_check returns TRUE for the ControlList. In the most simple form, the function checks whether ControlList is not empty. If the neighborhood of objects should only be investigated up to a certain “depth”, then an additional parameter for the number of steps that have to be performed can be used in the function condition_check. The control structure of the main loop works as follows: objects are selected from the ControlList, one at a time, and a similarity query is performed for this object. The procedures proc_1 and proc_2 perform some processing on the selected object as well as on the answers of the similarity query that will vary from task to task. Before repeating the loop, the ControlList is updated. Some or all of the answers which are not yet processed are simply inserted into the ControlList. The function filter(Answers, ...) removes from the set of answers at least those objects which have already been in the ControlList in previous states of the algorithm, if any exists. This must be done to guarantee the termination of the algorithm.

3.2 Instances of Iterative Neighborhood Exploration

In the following, we discuss several typical instances of the *ExploreNeighborhoods* scheme in order to show that many data mining applications follow this scheme:

- *Manual Data Exploration*

  When manually exploring a multimedia database, for example, in proc_2 the answers are visualized and the user may store the multimedia objects considered to be interesting. In the filter the user may prune answers which are too dissimilar from the initial start objects.

- *Spatial Association Rules*

  [KH 95] introduces spatial association rules describing associations between objects based on spatial neighborhood relations. For instance, a rule may be discovered stating that 80% of the selected towns are close to some water such as a lake or river. In this algorithm, the set StartObjects is equal to the set of all database objects of a specified type such as town. SimType corresponds to the type of spatial neighborhood such as *intersects* which is given by the user. proc_2 calculates the support, that is the relative frequency, of the retrieved pairs of objects and the filter passes all of these pairs which have at least a specified minimum support.

- *Density-Based Clustering*

  DBSCAN [EKSX 96] is a typical density-based clustering algorithm. To find a cluster, DBSCAN starts with some database object *o* and retrieves all objects density-reachable from *o* with respect to two parameters *Eps* and *MinPts*. Initially, ControlList contains an arbitrary database object and range queries with a range of *Eps* are used as similarity queries. proc_2 counts the answers and the filter passes all answers which have not yet been assigned to some cluster if the cardinality of the set of answers is at least *MinPts*.

- *Simultaneous Classification of a Set of Objects*

  In an astronomy database [Hog 97], for example, all new stars observed by a telescope during the night are processed and added to the database the next day. Part of this processing is to classify the set of new stars, that is to assign each of them to one of the well-known classes. *k*-nearest neighbor classifiers [Mit 97] are effective for this task and they issue a *k*-nearest neighbor query for each of the objects to be classified. In this case, proc_1 is empty. proc_2 finds the majority class in the set of *k*-nearest neighbors and filter always returns an empty list, that is no additional query objects are generated.

- *Spatial Trend Detection*

  A *spatial trend* has been defined as a regular change of one or more non-spatial attributes when moving away from a given start object *o* [EFKS 98]. Neighborhood paths starting from *o* model the movement and a regression analysis is performed on the respective attribute values for the objects of a neighborhood path to describe the regularity of change. For this data mining task, the *ExploreNeighborhoods* loop is additionally controlled by the number of steps (i.e. the length of a neighborhood path) and the procedures proc_1 and proc_2 perform the regression analysis on the paths.

- *Proximity Analysis*

  The goal of proximity analysis is to explain the existence of some cluster of objects by using the features of neighboring objects. [KN 96] presents an algorithm which can efficiently find the "top-*k*" objects that are "closest" to a given cluster. A second algorithm takes these *k* objects as input and finds the features that are common to most of them. Characteristic properties such as "most of the clusters are close to private schools and parks" may be discovered. In this case, StartObjects contains all the objects of the specified cluster. proc_2 considers the features of the *k*-nearest neighbors and returns the most common ones. The filter returns an empty list implying that no additional query objects are added.

3.3 Transformation into Multiple Query Form

An instance of Iterative Neighborhood Exploration can benefit from a multiple similarity query because the algorithmic scheme can be transformed into a multiple query form such that it uses multiple similarity queries instead of single similarity queries. If the evaluation of a multiple similarity query for m query objects can be performed more efficiently than the evaluation of the corresponding m single similarity queries - which is possible as we will see in the next sections - the runtime of the whole class of *ExploreNeighborhoods*-algorithms will be improved.

We assume the following method to be available:

```
ListOfAnswerSetsDB::multiple_similarity_query(ListOfObjects, SimTypes);
```

Let the similarity queries for all of the query objects in ListOfObjects be “somehow” performed simultaneously and let the generated answers for all of these queries be stored in some internal buffer of the DBMS. If each of the queries is completely answered after the call of multiple_similarity_query, successive calls containing queries which were already asked in a previous call of the method then can just pick the answers from the buffer.

Note, however, that we do not *require* the multiple similarity query to generate a complete set of answers for *each* of the posed queries. One call of a multiple similarity query must only guarantee that the answers for the first query object are complete. We will discuss this weak specification of a multiple similarity query in more detail in the next section. The intuitive meaning is that if we ask a similarity query for the first object, we can additionally inform the DBMS that the similarity queries for the other query objects will probably be asked later and the DBMS may use this information to improve the overall runtime for the set of queries by retrieving (some of) the respective answers in advance. The transformed algorithmic scheme called *ExploreNeighborhoodsMultiple* is presented in figure 3. As we can see, the reformulation can be done in a purely syntactical way. Thus, a query optimizer can automatically use multiple similarity queries to efficiently process an *ExploreNeighborhoods*-algorithm if a multiple similarity query is available as a basic DBMS operation.

Obviously, the algorithmic scheme *ExploreNeighborhoodsMultiple* performs exactly the same task as the original *ExploreNeighborhoods* scheme. The only differences are that a set of objects is selected from the ContolList instead of selecting a single object and a multiple similarity query is performed instead of a single similarity query. However, in one execution of the main loop, the algorithm processes only the first element of the set of selected objects and its corresponding set of answers.

```
ExploreNeighborhoodsMultiple(DB, StartObjects, SimTypes, ...)
  ControlList := StartObjects;
  while ( condition_check(ControlList, ...) = TRUE ) do
    ListOfObjects := ControlList.choose_multiple(...);  // ListOfObjects = [object_1, . . ., object_m]
    proc_1(ListOfObjects.first(), ...);
    SetOfAnswers:= DB.multiple_similarity_query(ListOfObjects, SimTypes);
        // SetOfAnswers=[answers_1, . . .,answers_m], SimTypes=[SimType_1, . . .,SimType_m]
    proc_2(SetOfAnswers.first(), ...);
    ControlList := (ControlList ∪ filter(SetOfAnswers.first(), ...)) - {ListOfObjects.first()};
  end while;
```

Figure 3: Algorithmic scheme *ExploreNeighborhoodsMultiple*

4 Multiple Similarity Queries

In this section, the notion of a multiple similarity query is presented in more detail.

Definition 4: (*multiple similarity query*)

Let $DB \subseteq Objects$ be a database containing n objects. Let $Queries = [Q_1, Q_2, ..., Q_m]$ be a sequence of m query objects $Q_i \in Objects$ and let $SimTypes = [T_1, ..., T_m]$ be the corresponding sequence of query types.

A *multiple similarity query*, denoted by *DB.multiple_similarity_query*(*Queries, SimTypes*), returns a sequence $Answers = [A_1, ..., A_m]$, containing for each element Q_i in *Queries* a corresponding set A_i of objects of *DB* where the following holds:

1.) $A_1 = DB.similarity_query(Q_1, T_1)$, and

2.) $A_i \subseteq DB.similarity_query(Q_i, T_i)$ for all $2 \leq i \leq m$.

Only for the first query, *all* answers must be determined in a single call of a multiple similarity query. The remaining queries may be answered completely or partially, depending on the implementation of the multiple similarity query. We will argue in the next subsection that an incremental implementation, i.e. only the first query is answered completely and other queries are answered partially in a single call of the method, may be more efficient if we consider the overall run-time of an *ExploreNeighborhoodsMultiple* algorithm.

Using multiple similarity queries instead of single similarity queries we may spend less I/O time and less CPU time for a set of queries. First, we read a single page only once for the whole set of queries. Second, knowing a whole set of query objects in advance, we can use the distances between these query objects to replace expensive distance computations by significantly cheaper distance comparisons using the triangle inequality.

Our algorithm for a multiple similarity query is depicted in figure 4. The only parts that differ from the algorithm for a single similarity query (see figure 1) - besides the obvious handling of *multiple* query objects and types - are as follows:

- restore_from_buffer([Q_1,..,Q_m],[T_1,..,T_m])
- buffer_answers([$Answers_1$,..,$Answers_m$])

  In the beginning, we have to restore (partial) answers from an internal buffer -if available- and we have to store generated answers into this buffer at the end.

- determine_relevant_data_pages([Q_1, ..., Q_m], [T_1, ..., T_m])

  This procedure returns the set of all data pages relevant for Q_1 and, additionally, it returns some or all of the relevant data pages for the remaining query objects, that is

$$relevant_pages(Q_1) \subseteq determine_relevant_pages([Q_1, \ldots, Q_m], [T_1, \ldots, T_m]) \subseteq \bigcup_{i=1}^{m} relevant_pages(Q_i)$$

- First, a subset of the set of all queries is chosen which should be completely answered. If the implementation is based on the linear scan, each data page is relevant. If using an index structure such as the X-tree, the set of all data pages which cannot be excluded from the search for at least one of the selected queries is determined from the directory of the tree. Note that our implementation of a multiple similarity query on top of an index structure converges to the method for the linear scan when the page selectivity of the index decreases, e.g. with increasing dimension of the data space. In the worst case, the index has no selectivity at all, which means

```
DB::multiple_similarity_query(objects [Q1, ..., Qm]; types [T1, ..., Tm])
    [Answers1, ..., Answersm] := Self.restore_from_buffer([Q1, ..., Qm], [T1, ..., Tm]);
    Self.determine_relevant_data_pages([Q1, ..., Qm], [T1, ..., Tm]);
    for i from 1 to m do QueryDistsi := Ti.Range;
    for i from 1 to m, for j from i+1 to m do QObjDistsij := dist(Qi, Qj);
    while Self.unprocessed_pages() do
        NextPage := Self.read_next_page_from_disk();
        for each object O in NextPage do
            for i from 1 to m do AvoidingDistsi := UNDEFINED;
            for i from 1 to m do
                if Self.page_is_relevant(NextPage, Qi) then
                    if not avoid_dist_computation(O,Qi,QObjDists,AvoidingDists) then
                        Distance := dist(O, Qi);
                        AvoidingDistsi := Distance;
                        if Distance ≤ QueryDistsi then
                            Answersi.insert(O);      // in ascending order of dist(A,Q)
                            if Answersi.cardinality() > Ti.Cardinality() then
                                Answersi.remove_last_element();
                                QueryDistsi := Self.adapt_query_dist(Distance,QueryDistsi,Ti);
                                Self.prune_pages(QueryDistsi);
    Self.buffer_answers([Answers1, ..., Answersm]);
    return [Answers1, ..., Answersm];
```

Figure 4: Algorithm *multiple_similarity_query*

that no data page can be excluded from the similarity search. The details of determining the relevant data pages are presented in section 5.1.

- avoid_dist_computation(O,Q_i,QObjDists,AvoidingDists)

  We calculate the inter-object distances for all pairs of query objects and store them into QObjDists. Distances which must be calculated are temporarily stored into AvoidingDists. These distances and the QObjDists are needed for the application of the triangle inequality performed by avoid_dist_computation(O,Q_i,QObjDists, AvoidingDists). The details of avoiding distance calculations are presented in section 5.2.

5 Efficient Support for Multiple Similarity Queries

In this section, we present techniques that significantly reduce the amount of disk I/O as well as the number of CPU operations needed to evaluate a multiple similarity query compared to a set of single similarity queries. Furthermore, we briefly discuss how to achieve a further performance gain when using parallelization techniques for the processing of multiple similarity queries. Note that there is an upper limit for the number *m* of multiple similarity queries which can be processed simultaneously. This limit is determined by the amount of main memory available to buffer the answers and by the computational overhead for calculating the inter-object

distances between all pairs of query objects. Therefore, we assume that a total number of $M \geq m$ similarity queries is processed in $\frac{M}{m}$ consecutive blocks of m multiple queries.

Let $C^i = C^i_{I/O} + C^i_{CPU}$ be the cost for simultaneously processing i similarity queries. Then, the cost for evaluating M queries using single similarity queries is equal to $M \times C^1$, the cost for evaluating M queries using multiple similarity queries is equal to $\frac{M}{m} \times C^m$. Consequently, for a multiple similarity query to improve the efficiency of single similarity queries, the following condition must hold: $C^m < m \times C^1$.

5.1 Reducing I/O Cost

We discuss the algorithm *multiple_similarity_query* from an I/O cost point of view for two different implementations - one on top of the linear scan and another one on top of an index structure such as the X-tree.

When performing a linear scan, the multiple similarity query performs a condition check for all query objects while performing a single scan over the database and returns a sequence of answers for each query object. If the dimension d of the data space is very high, the scan may actually be the most efficient method to answer similarity queries because, in general, the performance of index structures degenerates with increasing dimension d.

When using a tree-like index structure (e.g. X-tree) to answer a single similarity query, a set of data pages which cannot be excluded from the search is determined from the directory of the tree. These pages are then examined and the answers to the query are determined. To answer a multiple similarity query for a set Q = [Q_1, ..., Q_m] of query objects , we propose a similar procedure. First, we determine the data pages to be read as if answering only the similarity query for Q_1. However, when processing these pages, we do not only collect the answers in the neighborhood of Q_1 but we also collect answers for the Q_i (i=2, ..., m) if the pages loaded for Q_1 are also relevant for Q_i. After this first step, the query for Q_1 is completely finished and the answers for all the other objects are *partially* determined. To determine the complete answers for the other query objects [Q_2, ..., Q_m] we have to call the method repeatedly for [Q_2, ..., Q_m], [Q_3, ..., Q_m], ..., [Q_m]. However, in subsequent calls the partial answers are first restored from the internal buffer. For instance, the second call for [Q_2, ..., Q_m] will only consider data pages which are relevant for Q_2 but which have not been processed in the first call .

This *incremental processing* of a multiple similarity query has the advantage that (partial) answers to all of the queries can be presented to a user at a very early stage of the evaluation. Furthermore, the incremental approach is very efficient if an *ExploreNeigborhoods*-algorithm dynamically adds new query objects when processing the answers obtained for previous query objects. Let us assume a first call of DB.multiple_similarity_query([Q_1,...,Q_m], [T_1,..., T_m]). Furthermore, after this call let some answers A_1, ..., A_k for the query object Q_1 be inserted into the ControlList of the *ExploreNeigborhoods*-algorithm and let these objects be inserted into the sequence Q of query objects at the beginning of the second execution of the main loop. Then, the multiple similarity query is executed for Q = [Q_2, ..., Q_m, A_1, ..., A_k] implying that now all data pages are considered which have not been processed for object Q_1 but have to be loaded for object Q_2. It is very likely for an *ExploreNeighborhoods*-algorithm that some of these pages must also be considered for some of the objects A_i (i=1, ..., k). Then, the answers for the objects A_i are (partially) collected from the current data pages determined by the object Q_2. These pages will not be loaded again when A_i becomes the first element of Q. If we use a non-

incremental evaluation of a multiple similarity query we have to load these pages again, resulting in an overall higher number of disk I/Os.

For m multiple similarity queries $Q_1, \ldots, Q_m$ the I/O cost $C_{I/O}^m$ is proportional to $\left| \bigcup_{i=1}^{m} relevant_pages(Q_i) \right|$ where $|S|$ denotes the cardinality of a set S. Obviously, an I/O speed-up is achieved if (and only if) there are data pages which are relevant for more than one query object - more formally: if $\left| \bigcup_{i=1}^{m} relevant_pages(Q_i) \right| < \sum_{i=1}^{m} |relevant_pages(Q_i)|$.

In the case of the linear scan, it holds that $C_{I/O}^m = C_{I/O}^1$, because $relevant_pages(Q_1) = \ldots = relevant_pages(Q_m)$, and therefore, the condition $C_{I/O}^m < m \times C_{I/O}^1$ is obviously satisfied. The average I/O cost for one query object is $C_{I/O}^1 / m$ and the speed-up factor for a multiple similarity query compared to m single similarity queries (with respect to disk I/O) is exactly equal to m.

In the case of a tree-like index structure, the ratio $\sum_{i=1}^{m} |relevant_pages(Q_i)| / \left| \bigcup_{i=1}^{m} relevant_pages(Q_i) \right|$ which determines the actual speed-up factor cannot be analytically derived. However, in higher dimensions it is very likely that a data page is relevant for more than one query object, especially if the queries are dynamically generated by an *ExploreNeighborhoods*-algorithm. Therefore, we assume that the condition $C_{I/O}^m < m \times C_{I/O}^1$ is also satisfied in this case, even though we expect the gain of a multiple similarity query on top of a tree-like index to be smaller compared to an implementation for the sequential scan. Note that the performance of a multiple similarity query with respect to the I/O cost is never worse than the performance of a single query.

5.2 Reducing CPU Cost

The basic idea for reducing the CPU cost is to use the triangle inequality to avoid distance computations which are the most expensive operations when evaluating a similarity query. The proposed approach makes use of the fact that a distance calculation is typically much more expensive than a distance comparison. To apply the triangle inequality to avoid distance calculations, we need to know the distances for each pair of query objects (Q_i,Q_j) which have to be calculated and stored in advance. This computational overhead is (up to a certain value of m depending on the number n of database objects) relatively small compared to the savings of distance computations by such a preprocessing.

Intuitively, there are two cases where the calculation of $dist(Q_j,O)$ can be avoided for a query object Q_j and a database object O if we already know the distance between Q_j and a second query object Q_i and the distance $dist(Q_i,O)$ has already been calculated: first, the query objects Q_i and Q_j are close to each other and $dist(Q_i,O)$ is large; second, the query objects Q_i and Q_j have a large distance from each other and $dist(Q_i,O)$ is small.

To outline the proposed method more formally, we first define the notion of an avoidable distance calculation in the context of multiple similarity queries.

Definition 5: (*avoidable distance calculation*)
Let $Queries=[Q_1, ..., Q_m]$ be the query objects for a multiple similarity query, $O \in DB$, and let l, $1 \le l < m$, be a natural number. Furthermore, let the values of $dist(Q_i, Q_j)$ be known for all $1 \le i \le m$, $1 \le j \le m$, and let $dist(Q_i,O)$ be known for all $1 \le i \le l$. Let $QueryDist(Q_i)$, $1 \le i \le m$, denote the query distance of Q_i in a current execution step of the multiple similarity query. Then, we call the *calculation of* $dist(Q_{l+1},O)$ *avoidable* with respect to *Queries* if we can conclude that $dist(Q_{l+1}, O) \ge QueryDist(Q_{l+1})$ without having to calculate $dist(Q_{l+1},O)$.

To show that a distance calculation is avoidable, we apply the triangle inequality - satisfied by the metric distance function *dist* - to the triangle defined by two query objects Q_1 and Q_2 and a database object O. We obtain the following three inequalities which hold simultaneously:

(1) $dist(O, Q_1) \le dist(O, Q_2) + dist(Q_2, Q_1)$

(2) $dist(O, Q_2) \le dist(O, Q_1) + dist(Q_1, Q_2)$

(3) $dist(Q_1, Q_2) \le dist(Q_1, O) + dist(O, Q_2)$.

Inequality (1) can be used to show the avoidability of the calculation of $dist(Q_2,O)$ because it yields a lower bound for $dist(Q_2,O)$. This is formalized in the following lemma.

Lemma 1. Let $Q_1, Q_2 \in Objects$ be query objects and let $O \in Objects$ be a database object. Let *dist* be a metric distance function *dist*: $Objects \times Objects \rightarrow \Re^+$.
If $dist(O, Q_1) \ge dist(Q_2, Q_1) + QueryDist(Q_2)$ holds, then it follows that $dist(Q_2, O) \ge QueryDist(Q_2)$

Proof. We reformulate inequality (1) as follows: $dist(O, Q_2) \ge dist(O, Q_1) - dist(Q_2, Q_1)$. By assumption, $dist(O, Q_1) \ge dist(Q_2, Q_1) + QueryDist(Q_2)$. Then, $dist(O, Q_1) - dist(Q_2, Q_1) \ge QueryDist(Q_2)$. By exploiting the symmetry of *dist* we derive: $dist(Q_2, O) \ge dist(O, Q_1) - dist(Q_2, Q_1) \ge QueryDist(Q_2)$ ❑

Figure 5 (left) illustrates a situation where lemma 1 holds and the calculation of $dist(Q_2,O)$ can be avoided. Inequality (2) is not useful for the purpose of avoiding distance calculations because it yields an upper bound

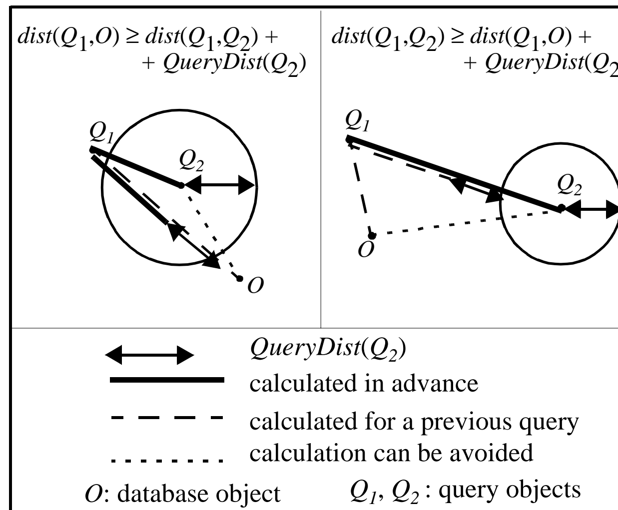

Figure 5: Illustration of lemma 1 and lemma 2

and not a lower bound for $dist(Q_2,O)$. Inequality (3), however, can be used analogously to inequality (1) and yields the following lemma.

Lemma 2. Let $Q_1, Q_2 \in Objects$ be query objects and let $O \in Objects$ be a database object. Let *dist* be a metric distance function *dist*: $Objects \times Objects \rightarrow \Re^+$.
If $dist(Q_2, Q_1) \geq dist(O, Q_1) + QueryDist(Q_2)$ holds, then it follows that $dist(Q_2, O) \geq QueryDist(Q_2)$.
Proof. Analogous to proof of lemma 1.

Figure 5 (right) depicts a case where lemma 2 can be applied to avoid the calculation of $dist(Q_2,O)$. To conclude the two above lemmata, figure 6 illustrates the area of database objects O for which the calculation of the distance from a query object Q_2 can be avoided. For example, the calculation of $dist(Q_2, O_1)$ can be avoided because $dist(O_1, Q_1) \leq dist(Q_2, Q_1) - QueryDist(Q_2)$ holds, and the calculation of $dist(Q_2,O_2)$ can also be avoided because $dist(O_2, Q_1) \geq dist(Q_2, Q_1) + QueryDist(Q_2)$ holds.

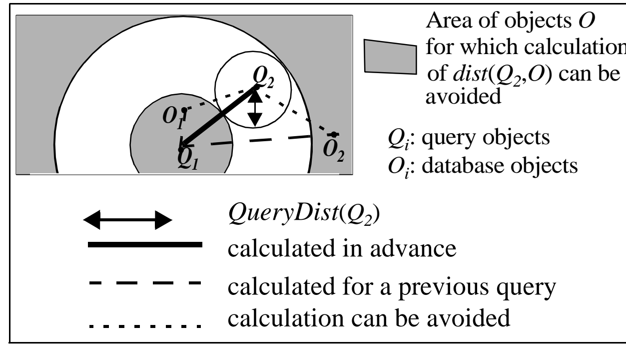

Figure 6: Area of database objects for which calculation of *dist* can be avoided

The CPU cost for processing m multiple similarity queries is given by the following formula

$$C_{CPU}^{m} = \frac{(m-1) \times m}{2} \times time(dist) + avoiding_tries \times time(comparison) + not_avoided \times time(dist)$$

where *avoiding_tries* denotes the number of (successful or not successful) applications of triangle inequalities and *not_avoided* denotes the number of distance calculations which actually have to be performed. Obviously, this formula contains several application dependent parameters which can only be determined experimentally. In the worst case, if no distance calculations can be avoided at all, it holds that $C_{CPU}^{m} > m \times C_{CPU}^{1}$. However, we observed that C_{CPU}^{m} is significantly smaller than $m \times C_{CPU}^{1}$ if m is small compared to the database size (see section 6 for details).

5.3 Potentials for Parallelization

In this section, we will briefly discuss the implementation of a multiple similarity query on top of a parallel query processor for a shared nothing environment. In such an environment, the data is distributed among s servers such that the same similarity query is performed on each server in parallel. However, each process has to

look only at its local part of the data which is s times smaller than the whole database. The communication overhead in this setting is very small so that a speed-up (compared to a sequential implementation) in the order of s can be expected, i.e. the cost C^m for performing m multiple similarity queries is reduced to $\frac{C^m}{s}$. The implementation of such a parallel query processor is trivial for the linear scan. For a parallel implementation, for example, of the X-tree see [Ber+ 97].

The transition from one computer to s computers of the same type also makes s-times the main memory available. If we use these additional resources when performing multiple similarity queries in parallel, we can gain a remarkable speed-up factor for $M \geq m \times s$ similarity queries which is larger than the number s of machines. This effect is due to the fact that we can increase the number m of query objects to be processed simultaneously if we have more memory to buffer the answers.

In a parallel environment each process produces only one s-th of the answer set for a query object on the average. Therefore, instead of evaluating M similarity queries in blocks of m queries on a single machine we can now use blocks of $m \times s$ queries. This means that the cost for evaluating M queries using parallel multiple similarity queries is equal to $\frac{M}{m \times s} \times \frac{C^{m \times s}}{s}$ compared to $\frac{M}{m} \times C^m$ for the sequential implementation. Consequently, the speed-up factor for a parallel multiple query versus a sequential multiple query is larger than s if $C^{m \times s} < s \times C^m$ holds. From sections 5.1 and 5.2 we know that we can expect this condition to be satisfied at least for the I/O cost. We cannot prove that the condition $C_{CPU}^{m \times s} < s \times C_{CPU}^m$ holds for the CPU cost but section 6.2 demonstrates this experimentally. Note, however, that even if this condition did not hold, we still would have the “normal” speed-up factor of s when using parallelization.

6 Performance Evaluation

We performed an extensive experimental evaluation of our technique for multiple similarity queries using real databases. The first database, part of the so-called *Tycho catalogue* [Hog 97], was provided by the European Space Agency (ESA) and contains 20-d feature vectors of 1,000,000 stars and galaxies. The second dataset was a large image database containing 64-d color histograms of 112,000 images from TV snapshots. We investigated two extreme instances of iterative neighborhood exploration discussed in section 3.2, one with independent queries and another one with highly dependent queries:

- On the Astronomy database, we tested *simultaneous classification of a set of objects*. M objects from the database were chosen randomly and a k-nearest neighbor query was performed for each of these query objects.
- On the image database, we simulated *manual data exploration* by a number of c concurrent users in the following way. We randomly selected a first query object for each of the users and performed a k-nearest neighbor query for each of them obtaining a total of $c \times k$ answers. Then we performed the following loop. While each of the hypothetic users chose one from his k current answers, for each of the current answers we prefetched their k-nearest neighbors. After restricting the set of answers to the answers of the objects chosen by the users, we continued the loop with these new query objects etc. Thus, in each loop we generated $m = c \times k$ new query objects for which we performed k-nearest neighbor queries.

We experimented with a broad range of k values and found that the average cost per k-nearest neighbor query was quite robust to the value of k. All the results reported in the following were obtained for $k = 10$ (Astronomy database) and $k = 20$ (image database) which are typical parameter values for the respective applications.

All experiments were performed on Intel Pentium II (300 MHz) based workstations running Linux 6.0, each workstation equipped with 128 MBytes of main memory. Both, the linear scan and the X-tree were implemented in C++. The block size of the X-tree was set to 32 KBytes and the buffer size was set to 10% of the X-tree size.

6.1 Reduction of I/O Cost

We begin by studying the effect of our technique for multiple similarity queries on the I/O cost. Figure 7 depicts the average I/O cost per similarity query with respect to the number m of multiple similarity queries for the Astronomy database as well as for the image database. For a single similarity query, the X-tree outperforms the linear scan by a factor of 4.5 and 3.1. For $m = 100$ query objects, however, the average I/O cost of the X-tree is 1.5 and 3.6 times the average I/O cost of the linear scan. While the enormous reduction of I/O cost (a factor of nearly m) is expected for the linear scan, it is worth noticing that also the average I/O cost of the X-tree is reduced by a factor of 8.7 and 15 for 100 multiple similarity queries.

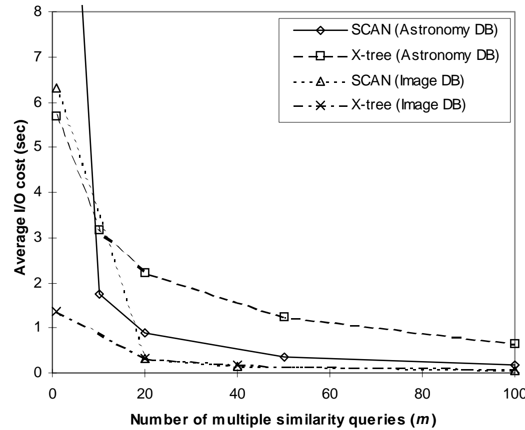

Figure 7: Average I/O cost per similarity query

6.2 Reduction of CPU Cost

The amount of CPU cost which can be saved when a data object is disqualified on the basis of the triangle inequality depends on the dimensionality of the database since the CPU cost for a distance calculation increases with the dimensionality whereas the CPU cost for evaluating the triangle inequality is constant. We measured the following average runtimes on our test databases. On 20-d data objects the CPU cost for calculating the Euclidean distance ($4{,}3\mu\text{sec}$) was 52 times the CPU cost for evaluating a triangle inequality ($0{,}082\mu\text{sec}$) and on the 64-d data objects the factor was 155 ($12{,}7\mu\text{sec}$ versus $0{,}082\mu\text{sec}$).

We measured the average CPU cost per query for 10, 20, 40, 50 and 100 multiple similarity queries (cf. figure 8). For the linear scan, the average CPU cost for a similarity query decreases from 4.3 sec to 0.6 sec on the Astronomy database when increasing m from 1 to 100. This corresponds to a reduction of the CPU cost by a factor of 7.1. On the image database, the factor of the CPU cost reduction is even 28. This effect can be ex-

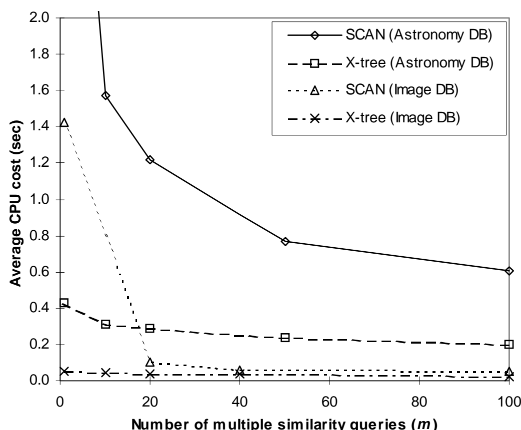

Figure 8: Average CPU cost per similarity query

plained when considering the distribution of the databases: the Astronomy database is almost uniformly distributed, the image database, however, is highly clustered. The linear scan profits from clustered databases for the following reason: if the distance computation for one data object from a cluster can be avoided it is likely that the distance computation for all other data objects lying in the same cluster can also be avoided.

For the X-tree, the effect of applying the triangle inequality is less than for the linear scan, it is 2.1 on the Astronomy database as well as on the image database. The reason for this smaller performance gain of the X-tree is the fact that due to its indexing properties, the X-tree solely investigates data objects which are close to query objects. Since data objects which have a large distance to the query objects - and therefore a high probability to be excluded from the distance calculation for most of the query objects - are not considered, the potential for CPU cost reduction is less than for the linear scan.

6.3 Reduction of the total query cost

We now consider the effect of our technique for multiple similarity queries on the total query cost and determine the achieved speed-up. For both databases, figure 9 shows the average total query cost as the sum of the average I/O cost and the average CPU cost. This can be done since the cost for managing the query process can

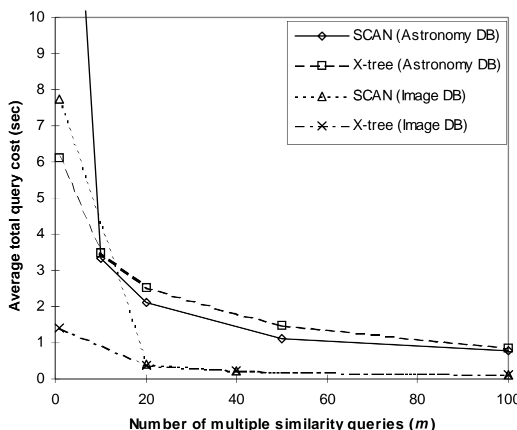

Figure 9: Average total query cost per similarity query

be neglected compared to the I/O cost and CPU cost. As expected, the average total query cost decreases with increasing m for the linear scan and the X-tree. An important observation we made is that for $m \geq 20$ (Astronomy database) and $m \geq 100$ (image database) the total query cost is dominated by the CPU cost when performing a linear scan. The average query cost of the X-tree was I/O bound for $m \leq 100$. Since the performance gain is higher for the linear scan, the linear scan outperforms the X-tree for $m \geq 10$ (Astronomy database) and $m \geq 100$ (image database).

Figure 10 depicts the corresponding speed-up. When comparing $m = 100$ to $m = 1$, the linear scan achieves a speed-up of 28 on the Astronomy database and 68 on the image database. For the X-tree, this speed-up is less due to the smaller benefits from the triangle inequality and the smaller reduction of I/O cost. However, we still observe a speed-up of 7.2 on the Astronomy database and 12.1 on the image database. Note that the speed-up factors are always higher on the image database. Similar to section 6.2, this effect can be explained with the distribution of the databases.

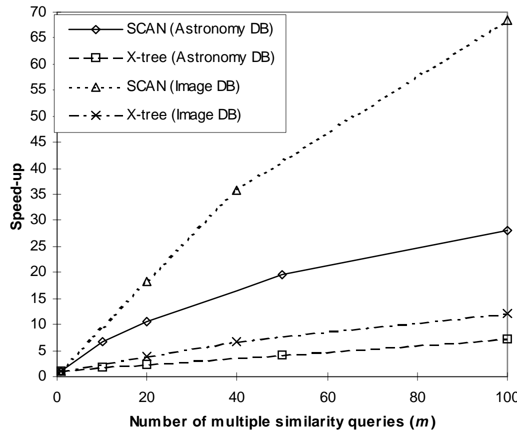

Figure 10: Speed-up with respect to *m*

6.4 Effects of parallelization

We also investigated the achievable speed-up when applying our technique for multiple similarity queries on top of a parallel query processor. The setting we used was a shared nothing environment with a TCP/IP network interconnecting 16 servers. For the implementation details of a parallel X-tree see [Ber+ 97]. For both databases, we performed $m = 100$ multiple k-nearest neighbor queries on a single server and while we increased the number of servers ($s = 4, 8, 16$) we proportionally increased m ($m = 400, 800, 1600$). Our technique of parallelization increases m in order to exploit the fact that s times the main memory becomes available (see section 5.3). Figure 11 depicts the achieved speed-up per similarity query comparing parallel multiple similarity queries to sequential multiple similarity queries.

On the Astronomy database, the parallel linear scan achieves a superlinear speed-up using up to 8 servers and a near linear speed-up of 13.4 using 16 servers. For larger server numbers, i.e. also larger numbers m of queries, two effects decrease the speed-up: (1) the cost for the computation of the distance for each pair of query objects and (2) the cost for applying the triangle inequalities for each database object which - in the worst case - is also quadratic in m. The second effect is less important for the X-tree because the X-tree visits only a considerably

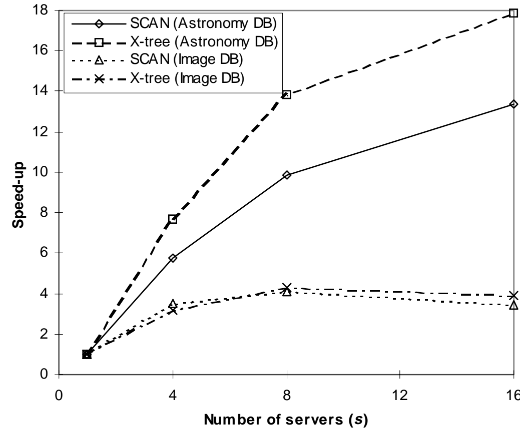

Figure 11: Parallelization Speed-up with respect to *s*

smaller number of database objects. Thus, the X-tree always achieves superlinear speed-up factors such as 17.9 for 16 servers.

On the image database, the achieved speed-up is sublinear, for example 4.1 (linear scan) and 4.3 (X-tree) for $s = 8$. Furthermore, we observe that the speed-up for the parallel linear scan as well as for the parallel X-tree using 16 servers is less than the speed-up using 8 servers. Again, this result is explained by the two above effects with a cost quadratic in m. Note that the influence of the initialization cost for the query distance matrix - which is independent from the database size - is much stronger here because the image database (112,000 objects) is significantly smaller than the Astronomy database (1,000,000 objects).

In figure 12 the overall speed-up of our technique is depicted with respect to s, i.e. the speed-up when performing multiple similarity queries on top of a parallel query processor compared to a sequential processing of single similarity queries. Thus, in figure 12 the combined effect of transformation into multiple queries and of parallelization is represented. On the Astronomy database, we observe an overall speed-up of 374 for the parallel linear scan and an overall speed-up of 128 for the parallel X-tree using 16 servers. On the image database, the overall speed-up factors using 8 servers are 279 for the parallel linear scan and 52 for the parallel X-tree.

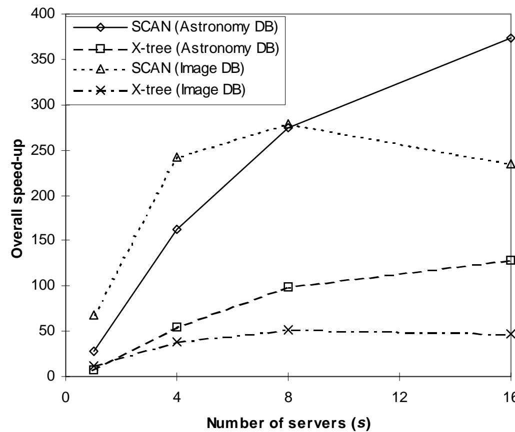

Figure 12: Overall Speed-up with respect to *s*

7 Conclusions

Similarity queries in the form of range queries and *k*-nearest neighbor queries are the most important query types in metric databases. Whereas in traditional query processing queries are issued independently, the typical scenario of many data mining applications is to explore the database by iteratively investigating the neighborhood of some start objects. In this paper, we propose a new query type, the so-called *multiple similarity query*, to speed-up such data mining applications by simultaneously processing sets of similarity queries. We introduced a generic scheme for many data mining algorithms and we developed a method to syntactically transform those algorithms in a way that they can use multiple similarity queries instead of single similarity queries. Our approach for efficiently processing multiple similarity queries includes two orthogonal techniques: first, the reduction of I/O cost by loading data pages only once and processing them for each query object. Second, the reduction of CPU cost by applying the triangle inequality in order to avoid expensive distance computations. Furthermore, we explored the potential of parallelization. The proposed techniques apply to any type of similarity query and to an implementation based on an index or using a sequential scan. An extensive experimental evaluation on real databases demonstrated the efficiency of our approach: by combining all of our techniques we achieved an overall speed-up with 16 servers in the order of 100 for an index-based implementation and in the order of 300 for an implementation using a sequential scan.

We argue that multiple similarity queries should be provided as a basic DBMS operation since they allow to speed-up the processing of many data mining algorithms. There are several directions to improve the efficiency of multiple similarity queries even more. We will investigate methods to reduce the initialization overhead implied by the query distance matrix. Furthermore, the potential of parallelization should be explored in more detail, e.g. the effects of various data declustering strategies.

References

[Ber+ 97] Berchtold S., Böhm C., Braunmüller B., Keim D. A., Kriegel H.-P.: "*Fast Parallel Similarity Search in Multimedia Databases*", Proc. ACM SIGMOD Int. Conf. on Management of Data, Tucson, AZ, 1997, pp. 1-12.

[BKK 96] Berchtold S., Keim D. A., Kriegel H.-P.: "*The X-Tree: An Index Structure for High-Dimensional Data*", Proc. 22th Int. Conf. on Very Large Data Bases, Bombay, India, 1996, pp. 28-39.

[BBKK 97] Berchtold S., Böhm C., Keim D. A., Kriegel H.-P.: "*A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space*", ACM PODS Symposium in Principles of Database Systems, 1997, Tucson, Arizona.

[BKSS 90] Beckmann N., Kriegel H.-P., Schneider R., Seeger B.: *"The R*-tree: An Efficient and Robust Access Method for Points and Rectangles"*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 322-331.

[CPZ 97] Ciaccia P., Patella M., Zezula P.: *"M-tree: An Efficient Access Method for Similarity Search in Metric Spaces"*, Proc. 23rd Int. Conf. on Very Large Data Bases, Athens, Greece, 1997, pp. 426-435.

[EFKS 98] Ester M., Frommelt A., Kriegel H-P., and Sander J.: *"Algorithms for Characterization and Trend Detection in Spatial Databases"*, Proc. 4th Int. Conf. on Knowledge Discovery and Data Mining, New York City, NY, 1998, pp. 44-50.

[EKSX 96] Ester M., Kriegel H.-P., Sander J., Xu X.: *"A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise"*, Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining, Portland, OR, 1996, pp. 226-231.

[Fal+ 94] Faloutsos C., Barber R., Flickner M., Hafner J., Niblack W., Petkovic D., Equitz W.: *'Efficient and Effective Querying by Image Content',* Journal of Intelligent Information Systems, Vol. 3, 1994, pp. 231-262.

[FPS 96] Fayyad U. M., .J., Piatetsky-Shapiro G., Smyth P.: *"From Data Mining to Knowledge Discovery: An Overview"*, in: Advances in Knowledge Discovery and Data Mining, AAAI Press, 1996, pp. 1 - 34.

[GG 98] Gaede V., Günther O.: "*Multidimensional Access Methods*", ACM Computing Surveys, Vol. 30, No. 2, 1998, pp. 170-231.

[Gut 84] Guttmann, R. *"R-trees: A dynamic index structure for spatial searching"*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, MA, 1984.

[Hog 97] Høg E. et al.: "*The Tycho Catalogue*“, Journal of Astronomy and Astrophysics, V.323, pp. L57-L60, 1997.

[HS 95] Hjaltason G.R., Samet H.: "*Ranking in Spatial Databases*", Proc. 4th Int. Symp. on Large Spatial Databases, Portland, ME, 1995, pp. 83-95.

[Kei 97] Keim D.: "*Efficient Support of Similarity Search in Spatial Data Bases*“, Habilitation Thesis, University of Munich, 1997.

[KH 95] Koperski K. and Han J.: "*Discovery of Spatial Association Rules in Geographic Information Databases*", Proc. 4th Int. Symp. on Large Spatial Databases, Portland, ME, 1995, pp. 47-66.

[KLTW 96] Keim D. A., Lee J. P., Thuraisinghaman B., Wittenbrink C.: "*Database Issues for Data Visualization: Supporting Interactive Database Exploration*", Proc. Workshop on Database Issues for Data Visualization, Atlanta, GA, 1995, in: Lecture Notes in Computer Science, Springer, 1996.

[KN 96] Knorr E.M. and Ng R.T.: "*Finding Aggregate Proximity Relationships and Commonalities in Spatial Data Mining*," IEEE Trans. on Knowledge and Data Engineering, Vol. 8, No. 6, Dec. 1996, pp 884-897.

[Mit 97] Mitchell T.M.: *"Machine Learning"*, McGraw-Hill, 1997.

[MJHS 96] Mombasher B., Jain N., Han E.-H., Srivastava J.: *"Web Mining: Pattern Discovery from World Wide Web Transactions"*, Technical Report 96-050, University of Minnesota, 1996.

[Sam 89] Samet H.: '*The Design and Analysis of Spatial Data Structures*', Addison-Wesley, 1989.

[SK 97] Seidl T., Kriegel H.-P.: "*Efficient User-Adaptable Similarity Search in Large Multimedia Databases*", Proc. 23rd Int. Conf. on Very Large Databases, Athens, Greece, 1997, pp. 506-515.

[WSB 98] Weber R., Schek Hans-J., Blott S.: "*A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces*", Proc. Int. Conf. on Very Large Databases, New York, NY, 1998, pp. 194-205.