EXHIBIT O

Efficient Similarity Search in Digital Libraries

Christian Böhm Bernhard Braunmüller Hans-Peter Kriegel Matthias Schubert

Computer Science Institute, University of Munich
E-mail: {boehm, braunmue, kriegel, schubera}@dbs.informatik.uni-muenchen.de

Abstract

Digital libraries are a core information technology. When the stored data is complex, e.g. high-resolution images or molecular protein structures, simple query types like the exact match query are hardly applicable. In such environments similarity queries, particularly range queries and k-nearest neighbor queries, turn out to be important query types. Numerous approaches have been proposed for the processing of similarity queries which mainly concentrate on highly dynamic data sets where insertion, update, and deletion operations permanently occur. However, only little effort has been devoted to the case of rather static data sets - a case that frequently occurs in digital libraries. In this paper, we introduce a novel technique for efficient similarity search on top of static or rarely changing data sets. In particular, we propose a special sorting order on the data objects which can be effectively exploited to substantially reduce the total query time of similarity queries. An extensive experimental evaluation with real-world data sets emphasizes the practical impact of our technique.

1. Introduction

In recent years, *digital libraries* have become a core information technology [10, 17]. Among the various important aspects of digital libraries the search for *similar* objects in the huge amount of digitized data has become an essential task. The QBIC system, for instance, contains a large image library which can be effectively searched for similar images by using *similarity queries* [22, 9]. The Brookhaven Protein Data Bank currently provides the atomic coordinates of several thousand proteins [2]. Here, the solution of the important molecular docking problem is supported by applying similarity queries on docking segments [19]. Another example are industrial CAD repositories which can effectively be used to reduce the cost of developing and producing new parts by maximizing the reuse of existing parts [21, 4]. Again, similarity queries are the most important query type in this process. Several other examples exist, e.g. geographical repositories, image collections and medical libraries.

The most common approach for efficient similarity search is to map data objects into some high-dimensional vector space (the so-called *feature space*). Similarity between two data objects is assumed to correspond to the distance of their feature vectors. Thus, searching for similar objects to a given query object is transformed into the problem of finding feature vectors which are *close* to the query feature vector. Popular examples of feature vectors are color histograms [8, 26], shape descriptors [15, 13], Fourier vectors [11, 1] and multi-parametric surface functions [18].

Typically, *range queries* and *k-nearest neighbor (k-nn) queries* are applied for the process of finding feature vectors which are close to a given query feature vector. Range queries retrieve all feature objects within a given radius ε from the query feature vector. For *k*-nn queries, the user provides a number *k* and receives the *k* feature vectors which are closest to the query vector. In general, a similarity query is a CPU and I/O intensive task and the conventional approach to address this problem is to use some multidimensional index structure [25, 12]. Unfortunately, even specialized index structures like the TV-tree [20] or the X-tree [6] often fail to process similarity queries efficiently when the dimensionality d of the feature space is too high ($d > 10$). However, this is a frequently encountered situation since the dimensionality of the feature space directly corresponds to the accuracy of the similarity search. In such environments, scan-based techniques like the VA-file [27] provide the most efficient query processing. These approaches mainly consider the case of highly dynamic environments with frequent insertion, update, and deletion operations. Only little effort has been devoted to the important case of static data sets. Obviously, static data sets provide more optimization potential than highly dynamic data sets. However, this optimization potential is not exploited by techniques which were developed for dynamic environments.

In this paper, we concentrate on the situation of static or rarely changing data sets (where the data set is known in advance) as they frequently occur in digital libraries. In particular, we introduce a novel technique (the *landmark file*) to significantly improve the query performance of scan approaches when storing static data sets. The main idea of the landmark file is to introduce a special order on the feature objects which can be exploited to substantially reduce the total amount of data which has to be scanned during the similarity search process.

The rest of our paper is organized as follows. In section 2, we review the problem of similarity search in the context of index-based and scan-based access methods. We present our approach in section 3 and section 4. We performed an extensive experimental evaluation in order to show the efficiency of our approach. The results are presented in section 5 and section 6 concludes the paper.

2. Similarity search in feature spaces

Range queries and *k*-nn queries are the most important similarity queries [7]. In the following, we formally define both query types.

Definition 1: *Range Query*

Let *DB* denote a set of feature vectors $v \in \Re^d$ and let $dist: \Re^d \times \Re^d \rightarrow \Re_0^+$ denote a distance function that measures the (dis-) similarity of two feature vectors $v_1, v_2 \in \Re^d$. Then, for a query feature vector $q \in \Re^d$ and a query range $\varepsilon \in \Re_0^+$, the range query returns the set

$$RQ(q, \varepsilon) = \{v \in DB \mid dist(q, v) \leq \varepsilon\}$$

Definition 2: *k-Nearest Neighbor Query*

For a query feature vector $q \in \Re^d$ and a query parameter $k \geq 1$, the *k*-nearest neighbor query returns the set $NN_q(k) \subseteq DB$ that contains *k* feature vectors from *DB*, and for which the following condition holds:

$$\forall v \in NN_q(k), \forall v' \in DB - NN_q(k) : dist(q, v) \leq dist(q, v')$$

Note that possibly several feature vectors exist which have the same distance to the query vector as the *k*-th feature vector in the answer set. In this case, the *k*-th feature vector in $NN_q(k)$ is a non-deterministic selection of one of those equally distanced feature vectors. Several distance functions for measuring the (dis-) similarity have been discussed. The Euclidean distance metric L_2, for instance, is one of the most frequently used similarity distance function, e.g. in the area of images, protein structures, CAD objects, or stock data.

Many sophisticated index structures have been proposed for efficiently processing similarity queries, but, only few techniques consider the case of static feature sets. In [24] a compaction technique for "packing" and reducing dead-space on R-trees [14] has been proposed for static pictorial feature sets. Other approaches follow the concept of bulk-loading when the feature set is completely known in advance. The Hilbert R-tree [16], for instance, uses the Hilbert space-filling curve to decompose the feature set into contiguous sequences which are stored in the data pages. In [5] a variant of the well-known Quicksort algorithm is used for a generic bulk loading method. Since index structures like the R-tree have been developed for low-dimensional feature spaces ($d < 5$) they offer only poor query performance for high-dimensional feature sets. In recent years, this problem has been addressed by developing high-dimensional index structures. The TV-tree [20], for example, uses so-called Telescope Vectors, i.e. feature vectors which may be dynamically shortened. The underlying assumption is that only dimensions with high variance are important for the query processing and therefore feature values of dimensions with low variance can be neglected. Another example is the X-tree [6] which uses the concept of directory supernodes. Whenever the split of a directory node would lead to a high overlap of the resulting nodes or to overlap minimal but extremely unbalanced nodes, the overflowing node is transformed into a supernode, i.e. a node with a larger than usual block size.

The main advantage of index-based access methods is the index selectivity during query processing, i.e. only a small fraction of the feature vectors has to be considered. This, however, induces costly random seek operations since the accessed index pages are generally not stored in contiguous disk blocks.

While these high-dimensional indexing techniques perform well for dimensions up to 10, even their performance often degenerates for higher dimensions. The reason for this effect is the so-called *curse of dimensionality*: Most of the measures one could define in a *d*-dimensional vector space, such as volume, area, or perimeter are exponentially dependent on the dimensionality of the space. Due to this effect, scan-based techniques, particularly the VA-file [27], turn out to provide better query performance when accessing high-dimensional feature spaces. The VA-file follows the idea of vector quantization. The feature space is divided into grid cells using α-quantiles in each dimension. Each feature vector is then represented by the address of the grid cell in which the feature vector lies. The main advantage of scan-based techniques is the sequential nature of their I/O operations which results in the absence of expensive random seeks. However, there is no selectivity in the query process involved and therefore all feature vectors have to be considered. Additionally, to the best of our knowledge, no optimization techniques have been proposed for scan-based techniques when the stored feature set is static.

3. The landmark file

As we have discussed in section 2, both query processing paradigms, indexing and sequentially scanning, have advantages and disadvantages. Our solution combines the advantages of both approaches and avoids their disadvantages. It adopts the idea of the scan, to read only from a single, contiguous interval of the file and avoids uncontrolled random seek operations. But our solution does not read and process the complete feature set. Thus, our solution inherits the selectivity from the indexing approach.

For static data sets, we can achieve this goal by keeping the feature set in a sort order according to an appropriate

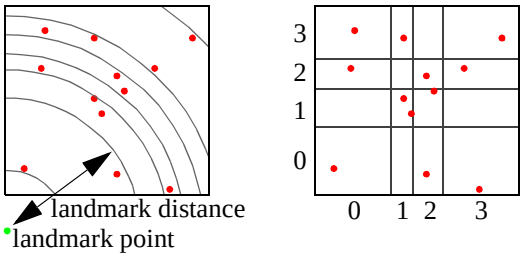

Figure 1. Landmark approx. (l.) and quantization (r.)

sorting key. For each query, the query processing algorithm determines the lower and upper bound of the sorting key and performs a sequential scan of the corresponding interval of the feature file. To choose the sorting key of the feature vectors, there are several possibilities, such as the projection to a single dimension. To assess the quality of a sorting key, recall the general objective: a substantial reduction of the number of scanned feature vectors. The interval into which the query is translated has to contain few feature vectors. In other words, the sorting key must distinguish as much as possible between different feature vectors. Obviously, the projection to a single dimension could fail to reach such a high distinguishing power. Therefore, our sorting key considers all relevant dimensions. A general approach is to use the Euclidean distance between a selected reference point and the feature vectors. If, for instance, the origin is selected as reference point, the sorting key corresponds to the sum of the squared dimension values. We will show in section 4, how an optimal reference point can be chosen.

3.1 Structure of the landmark file

In the following we show how the idea of sorting the feature vectors can be applied to establish selectivity in scan-based query processing. A point of the feature space (not necessarily contained in the data set) is chosen as reference point. This point is called the *landmark point* (cf. figure 1 left). The feature vectors are sorted in ascending distance to the landmark point (the sorting key is called the *landmark distance*).

We keep three representations of the feature vector file. In the *exact representation*, we store the full geometry of all feature vectors, i.e. the floating point values of the dimensions. In the *compressed representation*, we adopt the idea of the VA-file and store a quantized version of all feature vectors, i.e. the address (number) of the grid cell in which the corresponding feature vector lies (typically a few bits per dimension, cf. figure 1 right). The compressed representation requires substantially less space on secondary storage, and, therefore, the I/O cost compared to the exact representation is approximately reduced by the *compression factor*. On the other hand, the position of a feature vector is not exactly known such that a single access to the exact representation may be necessary.

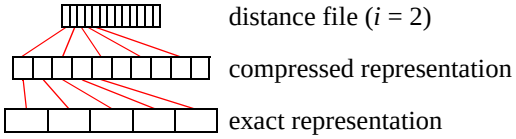

Figure 2. The landmark file

The third representation of the feature vector file is the *distance file*. This file does not contain information about each feature vector of the data set, but only for one vector out of i where i is a user-provided parameter, the *chunk parameter*. The landmark distance of each i-th vector of the ordered feature set is stored. This distance file partitions the feature space into spherical shells which contain a constant number i of feature vectors. The *landmark shells* for $i = 2$ are depicted in figure 1 left. There are no explicit pointers or links between the different representations of the files. As the sort order of the feature vectors is identical in all three representations, and since the length of each feature vector representation is constant, this reference is implicitly given by the position of the feature vector in the file (cf. figure 2).

Every query processing algorithm first considers the distance file and determines the landmark shells which potentially include query results. The corresponding part of the file with the compressed representation is scanned and candidates are collected. For every candidate which cannot be definitely classified as a query result by the compressed representation, a lookup to the exact representation is performed.

To process rare insertions, deletions, and updates we propose to store new or updated points in a separate file which is scanned sequentially without any optimizations for query processing. Whenever this overflow area becomes too large, the complete landmark file is reconstructed from scratch.

3.2 Query processing using landmarks

When the region of the query is exactly known in advance (such as in range queries), then query processing is straightforward:

- The distance file is loaded.
- Using the distance file, those shells are determined which are intersected by the query (cf. figure 3).
- For the intersected shells, the compressed representation is scanned.
- If necessary, the exact representation is accessed for

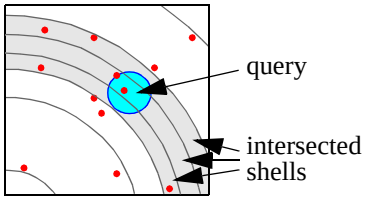

Figure 3. Range queries on the landmark file

each feature vector for which a decision cannot be made using the compressed representation only.

For (1-) nearest neighbor queries, the query region is not known in advance. Such queries are usually evaluated by increasing the radius of the query sphere until the nearest neighbor is covered by the query sphere. We adapt this algorithm for the landmark file (cf. figure 4):

- The distance file is loaded.
- Using the distance file, the shell containing the query vector is determined. This is the start shell.
- Beginning with the start shell, the unprocessed shell which is closest to the query vector is scanned successively. The scanned part of the compressed representation is extended, alternating at its upper and lower end. Whenever a feature vector is found which is closer than the current candidate, it is stored in a variable.
- This step is repeated until the distance between the next shell and the query vector is larger than the distance to the current candidate.

It is straightforward to extend this algorithm for *k*-nn search: Instead of a single candidate, the algorithm has to maintain a list with *k* candidates, and the distance to the last candidate is used as the stopping condition. For both query types, the algorithm must perform lookups to the exact representation in tie situations.

3.3 Parameter optimization

The chunk parameter *i*, which determines the size of the blocks for scanning in the nearest neighbor algorithm, is obviously a critical parameter. If it is chosen too small, the compressed representation is scanned in too small portions, which induces non-negligible I/O overhead. In contrast, if it is chosen too large, many compressed feature vectors may be scanned unnecessarily. To minimize the query processing time, the parameter *i* must be optimized.

Optimization problems like this are often solved by cost models such as [3]. In this special case, however, the estimation of the cost may be difficult because the intersection volumes between sphere shells and the query sphere must be computed. Therefore, we base our optimization on statistical information from previous runs of the algorithm or from tentative runs on a sample.

The information we are gathering is the means μ and the standard deviation σ of the number *a* of feature vectors with

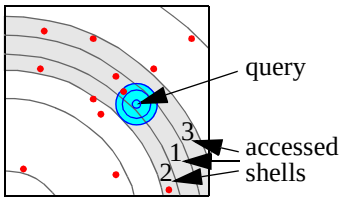

Figure 4. Nearest neighbor queries

a landmark distance in the range from $L(q) - r$ to $L(q) + r$, where $L(q)$ is the landmark distance of the query vector and r is the distance of the nearest neighbor of q. The number a corresponds to the number of vectors (= the number of shells) which must be processed by a query when $i = 1$.

For a known parameter a, we obtain the following cost function:

$$t(i, a) = (a + i) \cdot t_{tr} + \left(\frac{a}{i} + 1\right) \cdot (t_{pos} + t_{lat}) \qquad (1)$$

where t_{tr} is the transfer time per compressed feature vector, t_{pos} is the positioning time and t_{lat} is the latency time (rotational delay) of the disk drive. The first term of $t(i,a)$ indicates, that a compressed feature vectors are scanned during query processing, with an additional overhead of one block (i compressed feature vectors), because, on the average, half a block too much is scanned at the lower and the upper end of the scanned interval, respectively. In the second term of $t(i,a)$, the fraction $a/i + 1$ represents the number of separate I/O requests during query processing, for each of which a seek operation with arm positioning and rotational delay is required. This cost $t(i,a)$ can be minimized by setting the derivative to 0:

$$\frac{\partial}{\partial i} t(i, a) = 0 \Rightarrow i_{opt} = \pm\sqrt{a \cdot \frac{t_{pos} + t_{lat}}{t_{tr}}} \qquad (2)$$

Only the positive solution is valid, and it corresponds to a minimum, because the second derivative is (constantly) positive.

This solution optimizes the chunk parameter i for a known parameter a. However, this parameter is not constant for all queries. Therefore, we have to optimize i for a variety of different parameters a occurring in different queries. It turned out in our experiments that a is normally distributed. Under this assumption we can extend our model to optimize i for an a which is normally distributed with means μ and standard deviation σ:

$$\tilde{t}(i) = \int_0^\infty \left(\frac{1}{\sigma\sqrt{2\pi}} \mathbf{e}^{\frac{-(a-\mu)^2}{2\sigma^2}} \cdot t(i, a) \right) \mathbf{d}a \qquad (3)$$

This formula assigns to each cost $t(i,a)$ the probability of the corresponding a. The average over all possible a can be determined by integration with a ranging from 0 to infinity. In eq. (3), i can again be optimized by setting the derivative $\partial\tilde{t}(i)/\partial i$ to zero:

$$i_{opt} = \sqrt{\mu \cdot \frac{t_{pos} + t_{lat}}{t_{tr}}} \qquad (4)$$

The result is independent of the variance σ^2 and equal to the result of eq. (2) for a known a, where a is replaced by the means μ.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.