Nearest Neighbor Search on Vertically Partitioned High-Dimensional Data

Evangelos Dellis, Bernhard Seeger, and Akrivi Vlachou

Department of Mathematics and Computer Science, University of Marburg,
Hans-Meerwein-Straße, 35032 Germany
{dellis, seeger, vlachou}@mathematik.uni-marburg.de

Abstract. In this paper, we present a new approach to indexing multidimensional data that is particularly suitable for the efficient incremental processing of nearest neighbor queries. The basic idea is to use index-striping that vertically splits the data space into multiple low- and medium-dimensional data spaces. The data from each of these lower-dimensional subspaces is organized by using a standard multi-dimensional index structure. In order to perform incremental NN-queries on top of index-striping efficiently, we first develop an algorithm for merging the results received from the underlying indexes. Then, an accurate cost model relying on a power law is presented that determines an appropriate number of indexes. Moreover, we consider the problem of dimension assignment, where each dimension is assigned to a lower-dimensional subspace, such that the cost of nearest neighbor queries is minimized. Our experiments confirm the validity of our cost model and evaluate the performance of our approach.

1 Introduction

During the last decades, an increasing number of applications, such as medical imaging, molecular biology, multimedia and computer aided design, have emerged where a large amount of high dimensional data points have to be processed. Instead of exact match queries, these applications require an efficient support for similarity queries. Among these similarity queries, the *k*-nearest neighbor query (*k*-NN query), which delivers the *k* nearest points to a query point, is of particular importance for the applications mentioned above.

Different algorithms have been proposed [14, 16, 19] for supporting *k*-NN queries on multidimensional index-structures, like R-trees. Multidimensional indexing has extensively been examined in the past, see [12] for a survey of various techniques. The performance of these algorithms highly depends on the quality of the underlying index. The most serious problem of multi-dimensional index-structures is that they are not able to cope with a high number of dimensions. This disadvantage can be alleviated by applying dimensionality reduction techniques [3]. However, the reduced dimensionality still remains too high for most common index-structures like R-trees. After the high effort put into implementing R-trees in commercial database management systems (DBMS), it seems that their application scope is unfortunately quite limited to low-dimensional data.

A Min Tjoa and J. Trujillo (Eds.): DaWaK 2005, LNCS 3589, pp. 243 – 253, 2005.

In this paper, we revisit the problem of employing R-trees (or other multidimensional index-structures) for supporting *k*-NN queries in an iterative fashion. Our fundamental assumption is that high-dimensional real-world data sets are not independently and uniformly distributed. If this assumption does not hold, the problem is not manageable due to the well-known effects in high-dimensional space, see for example [24]. The validity of our assumptions allows the transformation of high-dimensional data into a lower dimensional space. There are, however, two serious problems with this approach. First, the dimensionality of the transformed data might still be too high, in order to manage the data with an R-tree efficiently. Second, this approach is feasible only when the most important dimensions are globally valid independent from the position of the query point. In our new approach, we address both of these problems by partitioning the data vertically and then indexing the (low-dimensional) data of each partition separately.

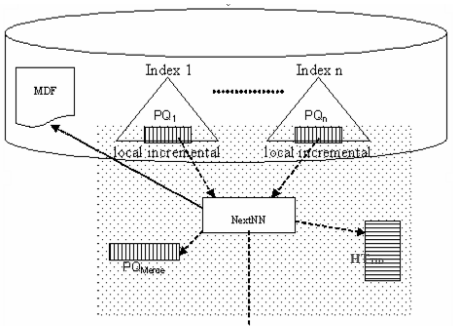

Fig. 1. Overview

Our basic architecture is outlined in Fig. 1. As mentioned above, the essential technique of our approach is to partition the data points vertically among different R-trees such that each dimension is assigned to one of them. Thus, the Cartesian product of the subspaces yields the original data space again. An incremental query is processed by running a local incremental query for each of the indexes concurrently. We keep the entire points in a separate multidimensional data file (*MDF*) on disk, where each point is accessible via a tuple identifier (*TID*). Additionally, a priority queue PQ_{Merge} and a hash table HT_{TID} are necessary for an efficient iterative processing of *k*-NN queries. The key problem that arises from our architecture is to determine an appropriate number of indexes for supporting *k*-NN queries efficiently. Thereafter, an effective assignment of the dimensions is indispensable. These problems are addressed in our paper. Moreover, we present different iterative algorithms for *k*-NN queries that employ the different indexes dynamically. Our approach is independent from the specific index-structure. We decide to use R-trees in the rest of the paper, simply because these structures are generally available and perform well in practice. For the sake of simplicity, we assume that the Euclidean metric L_2 is used as distance function throughout our paper.

The remaining paper is structured in the following way. In the next Section we discuss previous work. In Section 3, we present a multi-step nearest neighbor algorithm that dynamically exploits the information of internal nodes of the R-tree. Thereafter in Section 4, we provide a cost model relying on a power law and we propose a formula for computing the number of indexes. In Section 5 we present a dimension assignment algorithm. Our results of our experiments are presented in Section 6, and finally, we conclude in Section 7.

2 Related Work

Our work is closely related to the nearest neighbor algorithms that have been pro-posed in the context of multidimensional indexing [14, 16, 19]. Some of the proposals are dedicated to the problem of *k*-NN queries when *k* is known in advance, whereas others deal with the problem of distance scanning [14] also termed distance browsing [16] and incremental ranking where *k* is unknown and the query stops on demand. There are a large number of multidimensional index-structures that have been tailor-cut to supporting *k*-NN queries efficiently. Different filter-and-refine algorithms for *k*-NN queries were first presented in [18, 21]. These approaches, also known as Global Dimensionality Reduction (GDR) methods, are unable to handle datasets that are not globally correlated. In [7], a structure which is called Local Dimensionality Reduction (LDR) is presented that also exploits multiple indexes, one for each cluster. Unfortunately this strategy is not able to detect all the correlated clusters effectively, because it does not consider correlation nor dependency between the dimensions. Recently, there have been approaches where the high-dimensional points are mapped into a one-dimensional space [27, 28].

The origin of our work starts from [2] where a quite similar work has been pro-posed for supporting range queries on high-dimensional data. Based on a uniform cost model, the authors first present a formula for the optimal number of multi-dimensional indexes. Two techniques are then presented for assigning dimensions to indexes. The first one simply follows a round-robin strategy, whereas the other exploits the selectivity of the different dimensions such that the dimension with the highest selectivity is assigned to the first index and so on. The assignment strategy offers some deficiencies which makes them inadequate for *k*-NN queries. First, this strategy assumes dimensions being independently from each other. This may obviously lead to suboptimal assignments. Second, this strategy is even not applicable to high-dimensional data and *k*-NN queries. Since dimensions generally have the same domain, *k*-NN queries are symmetric in all dimensions and consequently, the selectivity is constant for all dimensions. Our approach is different from [2] as a new dimension assignment strategy is derived from the fractal dimension [1] of the data set. Moreover, instead of intersecting local results, a merge algorithm outputs the results in an iterative fashion.

Accurate cost models are important to our algorithms for decision making. We are primarily interested in cost models for estimating the cost for *k*-NN queries. There are many different cost models that have been developed for R-trees, such that [23]. The first models [24] were developed for uniformly and independently distributed data. Later, these models were extended to cope with non-uniform data. However, independency of dimensions is still assumed for the majority of the models, but unfortunately not satisfied for real data distributions, in general. The usage of the fractal dimension [17, 4] has lead to more accurate cost model since multidimensional data tend to behave in a fractal manner. Our approach employs such a cost model for automatically setting important parameters.

In parallel to the work on incremental ranking in the indexing community, there have been independently studies on a quite similar topic, so-called top-k queries in the area of multimedia databases [10, 11].

3 Incremental Nearest Neighbor Algorithms

In this section, we present three iterative algorithms for *k*-NN queries that comply with our architecture. Let *n* be the number of indexes, where each of them is responsible for a lower-dimensional subspace of the original data space. A global queue PQ_{Merge} is used for merging local results of the indexes. Since an index only delivers partial points, i.e. points from the subspaces, the multidimensional file (*MDF*) has to be accessed to obtain the entire point. We have developed the following three different incremental nearest neighbor algorithms, called Best-Fit, TA-Index and TA-Index$^+$.

Best-Fit: This algorithm is a straightforward adoption of the classical incremental algorithm [14, 16] for more than one index. During the initialization step of the query, the roots of all *n* indexes are inserted into PQ_{Merge}. In each iteration step, the algorithm pops the top element from PQ_{Merge}. If the top element is an index entry, the entry is expanded, i.e., the referenced page is read from the corresponding index and its entries (points) are added to PQ_{Merge}. If the top element is a partial point that is the first time on top, the entire point is read from *MDF* and inserted into PQ_{Merge}. In addition, we store its tuple identifier in the hash table HT_{TID}. We actually use HT_{TID} to check whether a partial point is the first time on top. If the top element is an entire point, we deliver the point as the next neighbor to the user. Note that partial points that appear more than once on top can be safely discarded.

TA-Index: This algorithm performs similar to TA [11], but supports arbitrary NN queries due to the fact that the indexes are able to deliver the data in an appropriate order. TA-Index performs different to Best-Fit as a local incremental NN query runs for every index concurrently. The partial points, which are delivered as the results of these local queries, are merged by inserting them into PQ_{Merge}. The algorithm performs similar to Best-Fit as partial points that appear on top are replaced by their entire points. In analogy to TA, we keep a partial threshold min_i for the i^{th} index where min_i is the distance between the last partial result of the i^{th} index and the query point of the NN query. An entire point from PQ_{Merge} is returned as the next nearest neighbor if its distance is below the global threshold that is defined as the squared sum of the partial thresholds. The faster the partial thresholds increase the higher the chance that the top element of PQ_{Merge} will become a result.

TA-Index$^+$: This algorithm is an optimized version of TA-Index as it additionally exploits the internal index entries to speed up the increase of the partial thresholds. Different to TA-Index is that the incremental NN queries on the local indexes are assumed to deliver not only the data points, but also the internal entries tries that are visited during processing.

```
nextNearestNeighbor()
while ( sqrt(min_1^2 + ... + min_n^2) < L2(q, PQMerge.top()) do
    ind = activityScheduler.next();
    cand = incNNPlus(ind).next();
    min_ind = L2,ind(q, cand);
    if ((cand is a point) and not
        HTTID.contains(cand.TID)) then
        obj = MDF.get(cand.TID);
        PQMerge.push(obj);
        HTTID.add(obj.TID);
return PQMerge.pop();
```

Fig. 2. Algorithm TA-Index+

Note that the distance of the delivered entries and points is continuously increasing. Therefore, the distances to index entries can also be used for updating min_i without threatening correctness. The incremental step of the algorithm is outlined in Fig. 2.

In order to improve the performance of the algorithms, it is important to reduce the number of candidates that have to be examined. The number largely depends on how fast the termination inequality (condition of the while-loop in Fig. 2) is satisfied in our algorithms. Therefore, we develop effective strategies for computing a so called *activity schedule* that determines in which order the indexes are visited. The goal of the activity schedule is a fast enlargement of the sum of the local thresholds min_i. The naive strategy is to use a round-robin schedule. This strategy however does not give priority to those indexes which are more beneficial for increasing the local thresholds. Similar to [13], we develop a heuristic based on the assumption that indexes that were beneficial in the recent past will also be beneficial in the near future.

Our activity schedule takes into account the current value of the partial threshold min_i as well as the number of candidates c_i an index has already delivered. The more candidates an index has delivered the less priority should be given to the index. Each time a candidate is received from an index, we increment the counter c_i. If the candidate has been already examined (by another index), we give an extra penalty to the i^{th} index by incrementing c_i once again. Our activity schedule then gives the control to the local incremental algorithm with maximum min_i/c_i. Note that our heuristic prevents undesirable situations where one index keeps control for a very long period.

In case of TA-Index$^+$ the distances to index entries influences not only the thresholds, but also the activity schedule. The internal nodes help to exploit the R-trees dynamically in a more beneficial way such that the number of examined candidates is reduced. This is an important advantage against the common TA [11].

The cost of our algorithm is expressed by a weighted sum of I/O and CPU cost. The I/O cost is basically expressed in the number of page accesses. IO_{index} refers to the number of accesses incurred from the local query processing, whereas IO_{cand} is the number of page accesses required to read the candidates from *MDF*. Note that IO_{cand} is equal to the total number of candidates if no buffer is used. The CPU cost consists of two parts. First, the processing cost for the temporary data structures like PQ_{Merge} and HT_{TID}. The more crucial part of the CPU cost might arise from the distance calculations.

4 A Cost Model Based on Power Law

We decided to use the fractal dimension as the basis for our cost model. The reason is twofold. First, there are many experimental studies, see [17], showing that multidimensional data sets obey a power law. Second, cost models for high-dimensional indexing, which go beyond the uniformity assumption, often employ a power law [4, 17]. Therefore, the fractal dimension seems to be a natural choice for designing a cost model.

In the following, we show that there is a strong relationship between the fractal dimension and the performance of nearest neighbor queries, especially in the case of multi-step algorithms. Let us assume that the data does not follow a uniform and in-

dependent distribution and that the query points follow the same distribution as the data points.

Given a set of points P with finite cardinality N embedded in a unit hypercube of dimension d and its fractal (correlation) dimension D_2, the average number of neighbors $\overline{nb}(r, D_2)$ of a point within a region of regular shape and radius r obeys the power law:

$$\overline{nb}(r, D_2) = (N-1) \cdot Vol(r)^{\frac{D_2}{d}},$$

where $Vol(r)$ is the volume of a shape (e.g. cube, sphere) of radius r, see [17]. The average Euclidean distance r of a data point to its k-th nearest neighbor is given by:

$$r = \frac{\left(\Gamma\left(1+\frac{d}{2}\right)\right)^{\frac{1}{d}}}{\sqrt{\pi}} \cdot \left(\frac{k}{N-1}\right)^{\frac{1}{D_2}}.$$

Since the ratio $\frac{k}{N-1}$ is always smaller than 1, an interpretation of the above formula reveals that a lower fractal dimension leads to a smaller average distance. A multi-step algorithm [21], where one index is used for indexing a subspace, is called optimal if exactly the points within this region are examined as candidates. The number of candidates that have to be examined by the optimal multi-step algorithm depends on the fractal dimension D_2' of the subspace, i.e. partial fractal dimension [25], with embedded dimensionality d', as shown in the following formula:

$$\overline{nb}(r, D_2') = (N-1) \cdot Vol(r)^{\frac{D_2'}{d'}} = (N-1) \cdot \frac{(\sqrt{\pi} \cdot r)^{D_2'}}{\Gamma\left(1+\frac{d'}{2}\right)^{\frac{D_2'}{d'}}}.$$

The above formula shows that a higher fractal dimension produces fewer candidates for the same radius. Thus, it is obvious that the performance of an optimal multi-step algorithm depends on the fractal dimension of the subspace that is indexed. In our multi-step algorithm, where more than one indexes are used, a region r_i of each index i, $(1 \le i \le n)$, such that $r = \sqrt{\sum_{1 \le i \le n} r_i^2}$ is dynamically exploited.

The average number of candidates for an index on a subspace with finite cardinality N, embedded in a unit hypercube of dimension d_i and fractal dimension $D_{2,i}$, is given by:

$$\overline{nb}(r_i, D_{2,i}) = (N-1) \cdot Vol(r)^{\frac{D_{2,i}}{d_i}} = (N-1) \cdot \frac{(\sqrt{\pi} \cdot r)^{D_{2,i}}}{\Gamma\left(1+\frac{d_i}{2}\right)^{\frac{D_{2,i}}{d_i}}}.$$

The total number of candidates is: $\overline{nb}(r) = \sum_{1 \le i \le n} \overline{nb}(r_i, D_{2,i})$.

In the above formula we include the duplicates that are eliminated during the process of our algorithm, but the number of duplicates is of minor importance for our cost

model. For the evaluation of this formula some assumptions are necessary. The goal of our cost model and dimension assignment algorithm is to establish a good performance for all indexes. Therefore, we assume that $d_i = \frac{d}{n} = d'$ and $D_{2,i} = D_2'$ for each index i. If this assumptions hold it is expected that the indexes are equally exploited based on the activity scheduler, thus $r_i = \frac{r}{\sqrt{n}} = r'$.

The I/O cost can be one page access per candidate in the worst case. As observed in [7] and as confirmed by our experiments this assumption is overly pessimistic. We, therefore, assume the I/O cost of the candidates to be the number of candidates divided by 2:

$$A_{cand}(r) = \frac{n}{2} \cdot (N-1) \cdot \frac{(\sqrt{\pi} \cdot r)^{D_2'}}{\Gamma\left(1+\frac{d'}{2}\right)^{\frac{D_2'}{d'}}}.$$

The I/O cost of processing a local k nearest neighbor query is equal to the cost of the equivalent range query of radius r_i. The expected number of page accesses for a range query can be determined by multiplying the access probability with the number of data pages $\frac{N}{C_{eff}}$, where C_{eff} the effective capacity. Based on the previous assumptions, the total cost of processing the local queries is:

$$A_{index}(r) = n \cdot \frac{N}{C_{eff}} \cdot \left(\sum_{0 \le j \le d'} \binom{d'}{j} \cdot \left(\left(1 - \frac{1}{C_{eff}}\right) \cdot \sqrt[D_2']{\frac{C_{eff}}{N-1}} \right)^j \cdot \frac{\sqrt{\pi^{d'-j}}}{\Gamma\left(\frac{d'-j}{2}+1\right)} \cdot (r')^{d'-j} \right)^{\frac{D_2'}{d'}}.$$

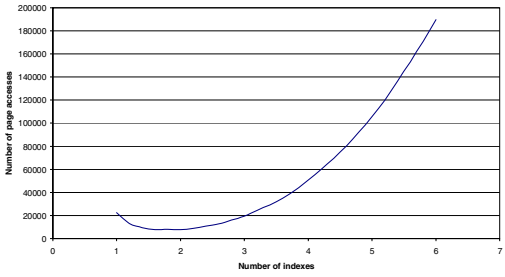

Fig. 3. Total Cost of k-NN query

The cost for processing a local nearest neighbor query within the lower-dimensional subspaces will increase with an increasing partial fractal dimension. Therefore our cost model defines the number of indexes where the sum of all I/O is minimized.

Fig. 3 shows the total cost over n indexes in a database consisting of 1,312,173 Fourier points in a 16-dimensional data space. The fractal dimension D_2 of the dataset is 10.56. We evaluate the parameter k by setting the expected average selectivity of a query as 0.01%. There is a clear minimum in n=2.

In our cost model we assume that the indexes currently maintain partial data points, whereas it might also be possible to keep the entire data points (in the leaves). This offers the advantage that *MDF* is not necessary anymore and therefore, A_{cand} = 0. However, the cost for processing the local incremental queries increase since, in case of the R-tree, the capacity of the leaves is reduced by a factor of n.

5 Dimension Assignment Algorithm

In this section, we sketch our algorithm for assigning d dimensions to n indexes, $n < d$. Based on the observation that subspaces with low fractal dimension produces more candidates, we follow two basic goals. First, each of the n indexes should have a high fractal dimension. Second, the fractal dimension should be equal for all indexes and therefore the indexes should perform similarly. These goals conform to the assumptions of our cost model and enhance the applicability of the cost model.

In general, an optimal solution of the dimension assignment problem is not feasible because of the large number of attributes. Instead, we employ a simple greedy heuristic that is inexpensive to compute while still producing near-optimal assignments.

A greedy algorithm is developed that starts with assigning the n attributes with the highest partial fractal dimension [25] to the indexes, where every index receives exactly one attribute. In the next iterations, an attribute, which has yet not being assigned to an index, is assigned to the index with minimum fractal dimension. The index receives the attribute that maximize the fractal dimension of the index. Notice that not all indexes have necessarily the same number of attributes.

Our algorithm is outlined in fig. 4. The function $D_2(S)$ calculates the fractal dimension of a subspace S. A_i represents the i^{th} attribute; $\boldsymbol{L}$ refers to the set of unassigned attributes, and da_i is the set of attributes assigned to the i^{th} index. The algorithm calls $O(d^2)$ times the box-counting algorithm [1] which calculates the fractal dimension in linear time w.r.t. the size of the dataset.

```
assignDimensions()
L = {A1,…,Ad};
for (i=1,…,n) do
    Amax = argmax A∈L { D2(A)};
    dadim = Amax ∪ dadim;
    L = L – {Amax};
while (L is not empty) do
    dim = argmin 1≤j≤n D2(daj);
    Amax = argmax A∈L D2(A ∪ dadim);
    dadim = Amax ∪ dadim;
    L = L – {Amax};
```

Fig. 4. Our Greedy Algorithm

6 Experimental Validation

For the experimental evaluation of our approach, indexes were created on two data sets from real-world applications. We set the page size to 4K and each dimension was represented by a double precision real number. Both data sets contain features of 68,040 photo images extracted from the Corel Draw database. The first dataset contains 16-dimensional points of image features, while the second dataset contains 32-dimensional points that represent color histograms. In all our experiments, we examined incremental k-NN queries where the distribution of the query points follows the distribution of the data points. All our measurements were averaged over 100 queries.

In order to illustrate the accuracy of our cost model, we show in Fig. 5. the estimated page accesses and the actual page accesses measured during the execution of *10*-NN queries by Best-Fit. The plot shows the I/O cost for both data sets where the fractal dimensions of the 16-dimensional and 32-dimensional datasets are 7.5 and 7.1, respectively. Note that the relative error is less than 20% in our experiments. Fur-

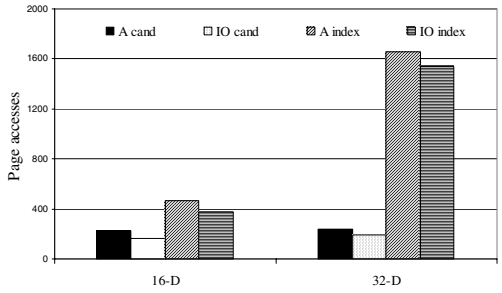

Fig. 5. Accuracy of our Cost Model

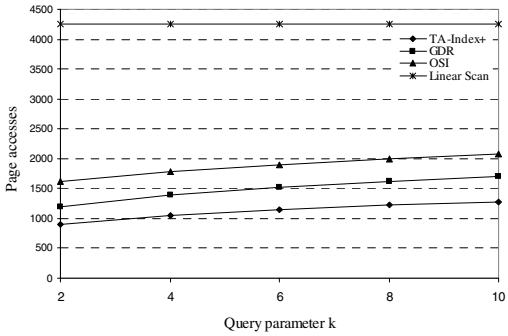

Fig. 6. Comparative Study

thermore, TA-Index$^+$ produces fewer page accesses, due to the usage of an activity schedule which is not considered by the cost model.

In the second experiment, we compare the following four techniques: TA-Index$^+$, Original Space Indexing (OSI), Global Dimensionality Reduction (GDR) and Linear Scan. We examined the I/O cost of *k*-NN queries for the 32-dimensional dataset. For TA-Index$^+$, we created two 16-dimensional R-trees as it was required from our cost model. Each of the corresponding subspaces has a fractal dimension of almost 3.75, resulting in well-performing R-trees. For the GDR method, we used the optimal multi-step algorithm on a 16-dimensional R-tree. The results of our experiment are plotted in Fig. 6. The plot contains four curves reporting the I/O as a function of *k*, where *k* denotes the desired number of neighbors. As illustrated, TA-Index$^+$ constantly outperforms the other techniques in our experiment.

7 Conclusions

The indexing problem for high-dimensional data is among the practically relevant research problems where solutions are difficult to achieve. The processing of nearest neighbor queries becomes only meaningful for high-dimensional data if we assume that there are high correlations in the data. We provide a new approach for querying vertically partitioned high-dimensional data. Three new algorithms for processing incremental nearest neighbor queries have been presented. In order to provide fast query processing, we addressed the problem of dimension assignment and provided approximate solutions based on the usage of the fractal dimension. An accurate cost model relying on a power law is presented. The results obtained by the experiments with real data sets consistently gave evidence that our technique is also indeed beneficial in practical scenarios.

References

1. Belussi A., Faloutsos C. Self-Spatial Join Selectivity Estimation Using Fractal Concepts. ACM Transactions on Information Systems (TOIS), Volume 16, Number 2, pp 161-201, 1998
2. Berchtold S., Böhm C., Keim D., Kriegel H-P., Xu X. Optimal multidimensional query processing using tree striping. Int. Conf. on Data Warehousing and Knowledge Discovery, 2000, pp 244-257.
3. Bingham E., Mannila H. Random projection in dimensionality reduction: applications to image and text data. Int. Conf. on Knowledge discovery and data mining, ACM SIGKDD, 2001, pp 245-250.
4. Böhm C. A cost model for query processing in high dimensional data spaces. ACM Transactions on Database Systems (TODS), Volume 25, Number 2, pp 129-178, 2000
5. Böhm C., Berchtold S., Keim D. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Computing Surveys, Vol. 33, pp 322-373, 2001
6. Böhm K., Mlivoncic M., Schek H.-J., Weber R. Fast Evaluation Techniques for Complex Similarity Queries. Int. Conf. on Very Large Databases (VLDB), 2001, pp 211-220.
7. Chakrabarti K. and Mehrotra S. Local Dimensionality Reduction: A New Approach to Indexing High Dimensional Spaces. Int. Conf. on Very Large Databases (VLDB), 2000, pp 89-100
8. Chaudhuri S., Gravano L. Evaluating Top-k Selection Queries. Int. Conf. on Very Large Databases (VLDB), 1999, pp 397-410.
9. Ciaccia P., Patella M., Zezula P. Processing complex similarity queries with distance-based access methods. Int. Conf. Extending Database Technology (EDBT), 1998, pp 9-23.
10. Fagin R., Kumar R., Sivakumar D. Efficient similarity search and classification via rank aggregation. ACM Symp. on Principles of Database Systems, 2003, pp 301-312.
11. Fagin R., Lotem A., Naor M. Optimal aggregation algorithms for middleware. Journal of Computer and System Sciences, Volume 66, Number 4, pages 614-656, 2003
12. Gaede V., and Gunther O. Multidimensional Access Methods. ACM Computing Surveys, Volume 30, pp 170-231, 1998
13. Güntzer U., Balker W-T., Kiessling W. Optimizing multi-feature queries in image databases. Int. Conf. on Very Large Databases (VLDB), 2000, pp 419-428.
14. Henrich A. A Distance Scan Algorithm for Spatial Access Structures. ACM-GIS, pp 136-143, 1994
15. Hinneburg A., Aggarwal C., Keim D. What Is the Nearest Neighbor in High Dimensional Spaces? Int. Conf. on Very Large Databases (VLDB), 2000, pp 506-515.
16. Hjaltason G. R., Samet H. Ranking in Spatial Databases. Advances in Spatial Databases, Int. Symp. on Large Spatial Databases, (SSD), LNCS 951, 1995, pp 83-95.
17. Korn F., Pagel B-U., Faloutsos C. On the ”Dimensionality Curse“ and the “Self-Similarity Blessing”. IEEE Transactions on Knowledge and Data Engineering, Vol. 13, No. 1, 2001.
18. Korn F., Sidiropoulos N., Faloutsos C., Siegel E., Protopapas Z. Fast nearest neighbor search in medical image databases. Int. Conf. on Very Large Databases (VLDB), 1996, pp 215-226.
19. Pagel B-U., Korn F., Faloutsos C. Deflating the Dimensionality Curse Using Multiple Fractal Dimensions. Int. Conf. on Data Engineering (ICDE), 2000, pp 589-598.
20. Roussopoulos N., Kelley S., Vincent F. Nearest neighbor queries. ACM SIGMOD, 1995, pp 71-79.

21. Seidl T., Kriegel H-P. Optimal Multi-Step k-Nearest Neighbor Search. ACM SIGMOD, 1998, pp 154-165.
22. Tao Y., Faloutsos C., Papadias D. The power-method: a comprehensive estimation technique for multi-dimensional queries. ACM CIKM, Information and Knowledge Management, 2003, pp 83-90.
23. Tao Y., Zhang J., Papadias D., Mamoulis N. An Efficient Cost Model for Optimization of Nearest Neighbor Search in Low and Medium Dimensional Spaces. IEEE TKDE, Vol. 16, No. 10, 2004.
24. Theodoridis Y., Sellis T. A Model for the Prediction of R-tree Performance. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 1996, pp 161-171.
25. Traina C. Jr., Traina A. J. M., Wu L., Faloutsos C. Fast Feature Selection Using Fractal Dimension. SBBD 2000, p.p. 158-171.
26. Weber R., Schek H-J., Blott S. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. Int. Conf. Very Large Databases (VLDB), 1998, pp 194-205.
27. Yu C., Bressan S., Ooi B. C., and Tan K-L. Quering high-dimensional data in single-dimensional space. The VLDB Journal, Vol. 13, pp 105-119, 2004.
28. Yu C., Ooi B. C., Tan K-L., and Jagadish H. V. Indexing the distance: an efficient method to KNN processing. Int. Conf. Very Large Databases (VLDB), 2001, pp 421-430.