

Dynamic Batch Nearest Neighbor Search in Video Retrieval

Jie Shao Zi Huang Heng Tao Shen Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland, Australia
{jshao, huang, shenht, zxf}@itee.uq.edu.au

Yijun Li
Nielsen Media Research
Australia
yijun.li@acnielsen.com.au

Abstract

To retrieve similar database videos to a query clip, each video is typically represented by a sequence of high-dimensional feature vectors. Given a query video containing m feature vectors, an independent Nearest Neighbor (NN) search for each feature vector is often first performed. Completing all the NN searches, an overall similarity is then computed, i.e., a single video retrieval usually involves the searches for m times. Since normally nearby feature vectors in a video are similar, a large number of expensive random disk accesses are expected to repeatedly occur, which crucially affects the overall query performance. Batch Nearest Neighbor (BNN) search is stated as a single operation that performs a batch of individual NN searches. This paper presents a novel approach to efficient high-dimensional BNN search called Dynamic Query Ordering (DQO) for advanced optimizations in both I/O and CPU cost. Observing the overlapped candidates (or search space) of a previous query may help to further reduce the candidate sets of succeeding queries, DQO aims to progressively find a query order such that the common candidates among queries are fully utilized to maximally reduce the total number of candidates. Modelling the candidate set relationship by a Candidate Overlapping Graph (COG), DQO iteratively selects the next query to be executed based on its estimated pruning power to the rest of queries with the dynamically updated COG. The extensive experiments show its significance.

1. Introduction

Recently the research on Content-based Video Retrieval (CBVR) has become very active. In a generic CBVR system, videos are first divided into a number of elemental segments, and the spatial and temporal features are then extracted, such as color, shape, texture and motion, etc. Normally, these features are represented by high-dimensional vectors, and each video is translated to a sequence of feature vectors. The high complexity of video features can be reduced to a level that can be efficiently managed by video summarization [2, 6]. The search module employs some high-dimensional access method [7, 3, 4] to speed up query processing.

In conventional similarity search, a query consumes a single NN search by traversing the indexing structure once. However, a distinguishing characteristic of video retrieval

is that, each video is described by a sequence of feature vectors, so as to the query. Denote a query clip as $Q = \{q_1, q_2, \dots, q_m\}$ and a database video as $P = \{p_1, p_2, \dots, p_n\}$, i.e., Q and P have m and n feature vectors (or representatives if some summarization is applied), to identify whether P is similar to Q or contains Q , typically for each $q_i \in Q$, a search is first performed in P to retrieve the similar feature vectors to q_i . A typical video similarity measure is to compute the percentage of similar feature vectors shared by two videos [2, 6, 5]. Given Q and P , their similarity is defined as:

$$Sim(Q, P) = \frac{\sum_{i=1}^m T(q_i)}{m},$$

where $T(q_i) = 1$ if q_i is relevant to some $p_j \in P$ and $T(q_i) = 0$ otherwise. Finding similar feature vectors is processed as a range or k NN search. The answers of all query vectors are then integrated to determine the final result.

Since there are m feature vectors in Q , totally the similarity search has to be performed for m time. For high-dimensional large video databases, it is a great challenge. We address this problem as Batch Nearest Neighbor (BNN) search which is defined as a single operation that performs a batch of individual NN searches on the same database simultaneously¹. Video retrieval is one of its applications. Given a query clip, a series of separated NN searches incurs a large number of random disk accesses thus most existing CBVR systems are constrained by testing on small video databases. The BNN search module introduced in this paper can enhance the efficiency of CBVR systems. In spirit, the goal of BNN search is somewhat similar to k NN join processing [8], which finds the NN (or k NN) for each point in one set from another set. However, k NN join deals with two large datasets which cannot fit into main memory, while the query set of BNN search is usually small e.g., the length of query video clip is relatively short, so that all the query points are resident in main memory. We assume that a maximal number of m queries can be processed in the batch, and at current state, BNN search is studied when the underlying access mechanism is not equipped with parallel processing capability.

We first generalize a query processing strategy Sharing Access (SA), which can be deployed with any arbitrary underlying access method to eliminate repeated random disk accesses. More importantly, since informed from a previ-

¹For the purpose of easy illustration, we use NN search ($k=1$ in k NN search) for discussion. As can be seen, the extension to range search and k NN search is straightforward.

ous query, the pruning conditions of succeeding queries may become tighter, a more advanced technique called Dynamic Query Ordering (DQO) is proposed to exploit the common candidates among queries to perform BNN search in an order such that the total number of unnecessary random candidate accesses is further reduced maximally. DQO progressively finds a query order by a Candidate Overlapping Graph (COG) towards minimizing the total cost. The triangle inequality can also be utilized to reduce distance computations for CPU optimization as well. To the best of our knowledge, no existing work has investigated the I/O cost further reduction by pruning condition tightening technique and dynamic ordering of query execution to optimize BNN search. Our experiments on real video datasets show the significant cost reduction is further achieved.

The rest of paper is organized as follows. Section 2 gives the preliminaries of our proposal. DQO is introduced in Section 3, followed by an extensive performance study in Section 4. Finally, we conclude in Section 5.

2. Preliminaries

Generally, the NN query processing contains two steps. First, data space is filtered to obtain some candidates by either a search radius typically in tree-based indexing structures, or a lower bound distance to query typically in scan-based approaches. For tree-based structures such as iDistance [3] which transforms each high-dimensional point to a one-dimensional distance value (with respect to a selected reference point) to be indexed by B^+ -tree, such candidates are all the points in the relevant data pages corresponding to the leaf nodes that intersect with search space. For the scan-based approaches such as VA-file [7] which divides the data space into 2^b rectangular cells and approximates data point that falls into the cell by a bit-string, such candidates are the points whose lower bound distances are not greater than the smallest upper bound distance determined so far. Then, the much smaller number of candidates are accessed from secondary memory and refined by computing the actual distances to query. The results are ranked for returning the NN. Though it is more clear to differentiate the terms of *random page accesses* used for tree-based indexing (I/O cost is gauged by the number of pages accessed) and *random candidate accesses* used for scan-based indexing (I/O cost is gauged by the number of candidates accessed), intrinsically they are similar. Note that I/O cost in VA-file comes from fetching both the approximation file and candidates into main memory. Since sequential I/O operation on the approximation file is much cheaper than random I/O operation on the candidates, and actually in BNN search, sequential I/O cost consumed in the filtering phase for multiple queries can be shared by loading the approximation file only once, we focus on reducing the random I/O cost.

Given a query clip $Q = \{q_1, q_2, \dots, q_m\}$, the naive way to find the NN of each q_i is to search separately, which is termed as *Sharing Nothing* (SN). According to the continuity of video sequences, consecutive queries describing the same shot/scene are usually close to each other in vector space and their search spaces are expected to overlap, and performing them in a batch usually leads to a much shorter overall running

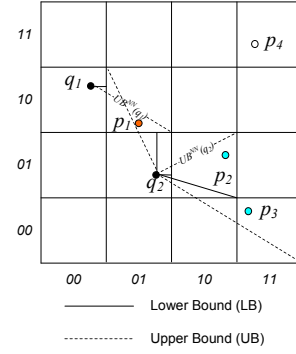


Figure 1. Repeated candidate accesses.

time than SN. Considering I/O cost of random disk accesses is quite expensive, it often becomes the bottleneck of query performance. The effect of optimization can be indicated by the number of random disk accesses saved.

If a candidate of current query will be re-accessed by some following queries, it is preferred to utilize this candidate already in main memory and directly apply distance computations. For instance, in the tree-based indexing iDistance, query composition introduced in our early work [6] analyzes the search ranges and composes the overlapped ranges into a single one to eliminate the duplicated accesses. The strategy introduced in [1] which loads a page once and immediately processes it for all the queries which consider it as a candidate page is also similar to this. For another instance, in the scan-based indexing VA-file, it is likely that some candidates can be avoided being accessed for multiple times. Since in BNN search, each $q_i \in Q$ is resident in main memory, the candidate sets of all queries can be merged together and only their union set needs to be loaded once. The common candidates are computed with multiple queries. Figure 1 is a 2-dimensional VA-file (each dimension with 2 bits) for illustration, where the smallest upper bound of each query, denoted as $UB^{NN}(q_i)$, is marked. As can be seen, the candidate set C_1 of q_1 includes p_1 only and the candidate set of q_2 includes p_1 , p_2 , and p_3 . First, the points in C_1 are accessed and processed (p_1 in Figure 1). For another query q_2 with $C_2 = \{p_1, p_2, p_3\}$, actual distance computations can be directly applied to the common candidates shared by C_1 and C_2 (p_1 in Figure 1). Through the merging of candidate sets, any duplicated candidate will be only accessed once.

The common essence between the query composition and merging operation strategies described above is generalized as *Sharing Access* (SA). SA loads the repeated candidates in a single pass for whole query set. Any potential redundancy can be eliminated by the sharing among individual NN determinations with a *union* operation on the candidates. The speeding up factor of SA r_{SA} is measured by:

$$r_{SA} = \sum_{i=1}^m |C_i| / |\bigcup_{i=1}^m C_i|,$$

where for tree-based indexing C_i is the considered page set of q_i and for scan-based indexing C_i is the candidate set of q_i . In the following, we adopt VA-file to illustrate actually the size of union can be further reduced for faster query response. Next, we show that a tighter pruning condition may be derived for succeeding query thus some unpromising candidates can be further identified and discarded directly.

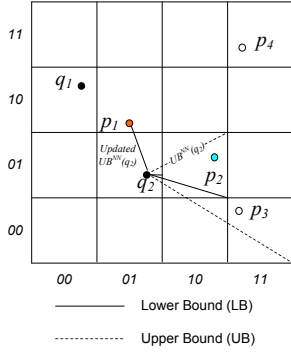


Figure 2. Pruning condition tightening.

3. Dynamic BNN Search

This section introduces our novel BNN search algorithm named Dynamic Query Ordering (DQO).

3.1. Pruning Condition Tightening

In VA-file, a tighter pruning condition (smaller $UB^{NN}(q_i)$) informed from a previous query can safely discard some false admits in the original candidates set, therefore, fully utilization of the common candidates may further reduce I/O cost. Re-consider Figure 2, interestingly, $d(q_2, p_1)$ is smaller than the original NN upper bound $UB^{NN}(q_2)$. Therefore it can be used as a new NN upper bound of q_2 , and random candidate accesses are only required for the updated C_2 which has excluded the points whose lower bounds are now greater than the updated pruning condition $d(q_2, p_1)$ (p_3 in Figure 2). Though the new candidate set becomes smaller, the correctness of query result is still guaranteed. Since some actually unpromising candidates are identified, the random accesses for them can be saved.

Fully unitizing the candidates available in main memory accessed by a previous query may lead the overall candidates for whole batch becomes only a subset of the union of all the candidate sets. Therefore, some random candidate accesses which still will be conducted according to SA are now confirmed as unpromising false admits and can be disqualified. We distinguish this exploration as *saving actually unpromising random candidate accesses with eliminating repeated random candidate accesses* described in SA.

3.2. Candidate Overlapping Graph (COG)

Since the progressive evaluation of smaller NN upper bound can further save I/O cost, we prefer an execution plan that benefits from the overall effect of pruning condition tightenings for all queries most. Compared with SA which loads the candidates corresponding to $\bigcup_{i=1}^m C_i$, the sequence of processing C_1, C_2, \dots, C_m now becomes crucial. Among all possible permutations of C_i , the optimal query ordering τ is a schedule of $C_{\tau_1}, C_{\tau_2}, \dots, C_{\tau_m}$ that minimizes the overall I/O cost by utilizing the overlapped candidates accessed by a previous query. To maximize the overall benefit from the partial candidate pre-fetching accumulated during each round of NN searches, we introduce a process of dynamically finding τ by Candidate Overlapping Graph (COG).

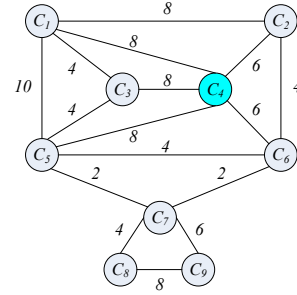


Figure 3. A candidate overlapping graph.

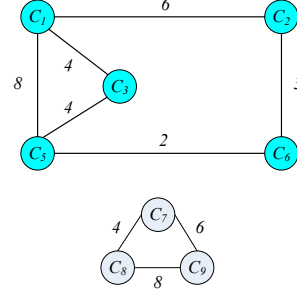


Figure 4. Updated candidate overlapping graph.

Definition 1 (Candidate Overlapping Graph). Given m queries with their , the overlapping relationship of their candidate sets is captured by their intersections. A candidate overlapping graph $G = \{V, E, \omega\}$ is a weighted graph representing the overlapped candidates. The vertex set $V = \{C_1, C_2, \dots, C_m\}$ is a set of candidate sets of individual queries. The edge set E is defined as: for each vertex pair C_i and C_j , there is an edge E_{ij} if $C_i \cap C_j \neq \emptyset$, and its weight ω_{ij} is the size of intersection $|C_i \cap C_j|$.

Figure 3 serves as an example of COG of 9 queries. The edges reflect how the two vertices they connect approximate to each other in vector space. Since some queries may be quite different from others, their candidate sets may not overlap with others. Therefore, some vertices may be isolated and it is possible that G is not a connected graph.

For COG, we have some observations. First, the potential pruning condition tightening only occurs between a vertex pair of a former query q_i and a latter query q_j where there is an edge E_{ij} connecting them (C_i and C_j have at least one common candidate). A tighter pruning condition can only be informed from the overlapped candidates of another query. Therefore, given a COG G , if a vertex C_i (and its associated edges) is removed, i.e., q_i is processed, only the vertices (and their associated edges) connecting to C_i are possible to be reduced. Figure 4 shows an example of removing C_4 from Figure 3, where the shaded vertices and their edges are possible to be updated. Note that in Figure 4, due to updating C_5 and C_6 , some of their candidates are excluded and they become sharing no common candidates with C_7 now. Thus, COG may become a unconnected graph. A query in an isolated subgraph of G can never contribute to or informed by the pruning condition tightening of a query in another isolated subgraph, i.e., there is no interaction of smaller NN upper bound deriving. Therefore, given G , if its each maximum connected subgraph G' gets its own optimal query order τ' , then serially executing each τ' in any arbitrary order is an optimal query order τ .

3.3. Dynamic Query Ordering (DQO)

The framework of Dynamic Query Ordering is presented below. DQO is an iterative process consisting three steps:

- **Select a vertex:** A vertex C_i is dynamically selected from the iteratively updated graph G for execution. This step determines the query order. Finding an optimal query order τ is a combinational optimization problem since whether the pruning condition can be tightened for each query can not be anticipated. Thus, effective heuristic methods are needed to find near-optimal orders. A general guideline is to process the queries expected to maximally reduce the other candidate sets earlier. In this way, the NN upper bound $UB^{NN}(q_i)$ will approach to the actual NN distance $dist^{NN}(q_i)$ faster.
- **Execute the query:** Once a vertex is selected, its candidates are randomly accessed for actual distance computations. This step determines the I/O cost. The more candidate set size is reduced, the less I/O cost.
- **Update G :** Once q_i has been executed, the vertices connecting to C_i can be updated, so as to their associated edges. To do so, for each C_j with $\omega_{ij} \neq 0$, the minimal distance between q_j and its overlapped candidates in C_i , i.e., $\min(d(q_j, p_i), \forall p_i \in C_i \cap C_j)$, is computed. If this minimal distance is smaller than the current NN upper bound $UB^{NN}(q_j)$, then update $UB^{NN}(q_j) = \min(d(q_j, p_i), \forall p_i \in C_i \cap C_j)$. All the candidates in C_j whose lower bounds are now greater than the smaller $UB^{NN}(q_j)$ can be pruned safely, thus $|C_j|$ is reduced. Correspondingly, the weights of edges connecting to C_j are also updated.

The process is repeated until all the vertices have been processed and their NN results are returned. When G becomes not a connected graph, the algorithm applies to its each maximum connected subgraph. The query order determined in the first step affects the degree of candidate set reduction in the third step. We propose two efficient heuristic methods to find near-optimal orders.

Since the future pruning condition and actual candidate reduction size are unpredictable, utilizing available information at current stage is reasonable and promising. Processing a query only affects its connecting neighbors. Our first intuition is that a query order maximizing the total volume of overlap in the neighborhoods of consecutive queries may reduce the candidates most. This leads to our first heuristic for query selection.

Heuristic 1. Given a COG G of m queries, a vertex C_i is selected to be searched next if its $\sum_{j=1}^m \omega_{ij}$ ($j \neq i$) is maximal.

Heuristic 1 suggests to greedily select a vertex with the largest number of total overlapped candidates with others as a pioneer. More common candidates not only mean more repeated random candidate accesses can be shared with its neighbors as in SA, but also imply a larger chance to derive tighter pruning conditions. In G , to process such a more influential query is equal to select the vertex with the maximal total

weights of associated edges². Recall the example in Figure 3, this vertex corresponds to C_4 . After processing q_4 , all the associated edges are removed and the sizes of its neighboring vertices are consequently reduced, as shown in Figure 4.

Heuristics 1 assumes the volume of overlap is proportional to the volume of candidates pruned. However, in real situation, after C_i is processed, the size of C_j itself also affects the number of candidates to be pruned. Meanwhile, if there are more candidates in C_j whose lower bounds are greater than the lower bound of an overlapped candidate, more candidates are potentially pruned from C_j . That is, the relative position of an overlapped candidate in C_j ranked by lower bound also affects the number of candidates to be pruned. Taking them into consideration, we have the second heuristic.

Heuristic 2. Given a COG G of m queries, a vertex C_i is selected to be searched next if its $\sum_j \sum_k (|C_j| - \text{Position}(p_k, C_j))$ is maximal, where C_j is a neighboring vertex of C_i , $p_k \in C_i \cap C_j$ and function $\text{Position}(p_k, C_j)$ returns the relative position of p_k in C_j ranked by lower bound in the ascending order.

$|C_j| - \text{Position}(p_k, C_j)$ is the anticipated maximal number of candidates pruned by p_k in C_j , i.e., all the candidates having greater lower bounds than that of p_k would be pruned. Thus in Heuristics 2, a query is selected based on its maximal pruning power. The performance of both heuristics will be studied in experiments.

3.4. CPU Optimization

CPU cost can also be further optimized, because not all the common candidates contributes to the pruning condition tightening. Triangle inequality can be employed to avoid the actually unnecessary computations on some helpless overlapped candidates. Given two queries q_i and q_j and $p_k \in C_i \cap C_j$, assume q_i is processed, if $|d(q_i, q_j) - d(q_i, p_k)| \geq UB^{NN}(q_j)$, since $d(q_j, p_k) \geq |d(q_i, q_j) - d(q_i, p_k)|$, we have $d(q_j, p_k) \geq UB^{NN}(q_j)$. Thus, some overlapped candidates which cannot tighten the pruning condition can be avoided from actual distance computations.

4. Experiments

We test on a real video database consisting of about 3,000 TV commercial clips. They are recorded at PAL frame rate of 25fps. The time length for each clip is about 15 seconds, i.e., the database consists of about 1,100,000 frames. Four feature datasets in 8-, 16-, 32- and 64-dimensional RGB color spaces were generated. All the results reported are the average based on 20 query clips selected from the database. Each query clip consumes a BNN search with about 375 frames. By default, we set $k=100$ in k NN search, i.e., each frame searches for 100 most similar frames from the database. To clearly see the improvement achieved by DQO, we use SA as the baseline for comparison, and denote DQO using Heuristic 1 and Heuristic 2 as DQO1 and DQO2, respectively. We define *Improvement Ratio* (IR) as the ratio of the total number of candidates

²When there is a tie, select the vertex whose size is smaller.

Bit number per dimension	6	7	8	9
Average number of candidates	2338	510	196	140

Table 1. Bit number per dimension vs. average number of candidates.

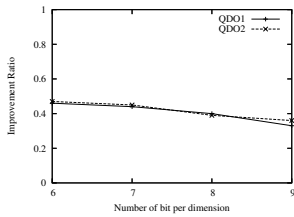


Figure 5. IR vs. bit number per dimension.

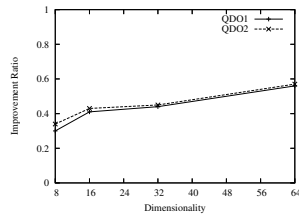


Figure 6. IR vs. dimensionality.

saved by DQO over the total number of candidates in SA. It shows the percentage of candidates further pruned by DQO.

We first test the effect of bit number for encoding each dimension in VA-file using the 32-dimensional dataset. The average number of candidates per query frame is shown in Table 1. As expected, as more bits are used for each dimension, the number of candidates for each query frame drops quickly.

Figure 5 shows the effectiveness of DQO1 and DQO2 for different bit numbers per dimension. Observed from Figure 5, our DQO further improves SA by pruning away more than 30%-50% candidates. This confirms the effectiveness of DQO. Also, the IRs of both DQO1 and DQO2 are reduced gradually as the bit number per dimension increases. This is reasonable since the number of candidates becomes smaller as the bit number per dimension increases (Table 1), so as to the number of overlapped candidates. Moreover, although DQO2 performs better than DQO1, such improvement is very marginal. This is probably due to nearly uniform distribution of overlapped candidates in most candidate sets. For the rest of experiments, we use 7 bits per dimension.

Next, we test the effect of dimensionality on Improvement Ratio with four feature datasets. As shown in Figure 6, again DQO improves SA greatly. Very interestingly, as dimensionality goes up, DQO performs better. A small tightening in the pruning condition is expected to prune away more candidates in a higher dimensional space. As we will also see later in Figure , more overlapped candidates are potentially used to further tighten the pruning condition in a higher dimensional space. This is another reason for the up trends in Figure 6.

It is straightforward to extend DQO to k NN search by replacing $UB^{NN}(q_i)$ with $UB^{kNN}(q_i)$ as the pruning condition. We test the effect of k on IR using the 32-dimensional dataset. Figure 7 indicates that DQO is slightly more effective for a larger k . The reason is that a larger k corresponds to larger candidate sets. The chance of a larger $UB^{kNN}(q_i)$ to be tightened is higher.

We also test how many percentage of distance computations for overlapped candidates can be saved by CPU optimization. Here Improvement Ratio (IR) is re-defined as *the ratio of the total number of distance computations saved with CPU optimization over the total number of distance computations without CPU optimization for overlapped candidates*

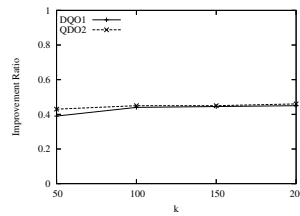


Figure 7. IR vs. k .

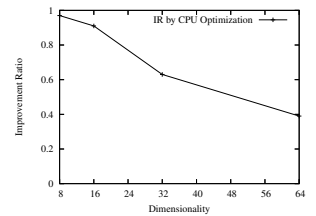


Figure 8. IR by CPU optimization.

in DQO. Four datasets are used and results are shown in Figure 8. The CPU optimization further reduces the number of distance computations significantly. When the dimensionality is relatively small (<16), it can save more than 95% distance computations. As the dimensionality increases, IR drops quickly. This indicates less number of overlapped candidates can be avoided for full distance computations in a higher dimensional space, i.e., more overlapped candidates can help to further tighten the pruning condition.

5. Conclusions

This paper proposes a Dynamic Query Ordering strategy to facilitate batch-oriented similarity query processing in high-dimensional space. How to progressively choose an optimal execution order is exploited as the key to speed up BNN search. Experiments on real video datasets show that the proposed methods outperform SA execution measured by the number of random candidate accesses, and the CPU cost can also be further reduced based on triangle inequality.

Acknowledgment: The work reported in this paper has been supported by Australian Research Council under grant DP0663272.

References

- [1] B. Braunmüller, M. Ester, H.-P. Kriegel, and J. Sander. Efficiently supporting multiple similarity queries for mining in metric databases. In *ICDE*, pages 256–267, 2000.
- [2] S.-C. S. Cheung and A. Zakhor. Efficient video similarity measurement with video signature. *IEEE Trans. Circuits Syst. Video Techn.*, 13(1):59–74, 2003.
- [3] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive b^+ -tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.
- [4] N. Koudas, B. C. Ooi, H. T. Shen, and A. K. H. Tung. Ldc: Enabling search by partial distance in a hyper-dimensional space. In *ICDE*, pages 6–17, 2004.
- [5] H. Lu, B. C. Ooi, H. T. Shen, and X. Xue. Hierarchical indexing structure for efficient similarity search in video retrieval. *IEEE Trans. Knowl. Data Eng.*, 18(11):1544–1559, 2006.
- [6] H. T. Shen, B. C. Ooi, X. Zhou, and Z. Huang. Towards effective indexing for very large video sequence database. In *SIGMOD Conference*, pages 730–741, 2005.
- [7] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.
- [8] C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: An efficient method for knn join processing. In *VLDB*, pages 756–767, 2004.