Large-Scale Similarity-Based Join Processing in Multimedia Databases

Harald Kosch and Andreas Wölfl

University of Passau, Germany
Distributed Information Systems, Faculty of Informatics and Mathematics
{Harald.Kosch,Andreas.Woelfl}@uni-passau.de

Abstract. This paper presents efficient parallelization strategies for processing large-scale multimedia database operations. These strategies are implemented by extending and parallelizing the GiST (Generalized Search Tree)-framework. Both data and pipeline parallelism strategies are used to execute multi join operations. We integrate the parallelized framework into an Oracle 11g Multimedia Database using its extension mechanisms. Our strategies and their implementations are tested and validated with real and random data sets consisting of up-to 10 millions of image objects.

Keywords: Multimedia Databases, Similarity-based Operations, Parallel Processing.

1 Motivation

In multimedia databases, the set of multimedia objects are described by a collection of features. In the case of images, examples of features include color histograms, color moments, textures, shape descriptions and so on [1]. Common multimedia database operations are *similarity-based selection queries* [2]. Two main types of selections are considered. First, one looks for objects whose feature vectors are within a given range (range queries) to the feature vector of a given query object. Second, one finds objects whose feature vectors have the most similar values to the feature vector of a given query object (nearest-neighbor queries). In addition, *similarity-based join queries* are considered [3,4,5]. The similarity-based join is useful to merge two sets of multimedia objects based on their pairwise similarity. It can be employed within a single database, for instance in social media applications to compute pairwise similarity of images posted by different interest groups. Similarity-based joins may also be employed to merge sets of image objects stemming from different repositories in order to find out pairwise similarity and thus create interlinking among them. In this scope, our paper studies methods for efficiently joining large sets of image objects. Our solution strategies include several data and pipeline parallelization methods.

2 Multimedia Database Operations

Let M_1 and M_2 be two image tables in a multimedia database. The image tables contain the image objects together with their feature vectors extracted beforehand from the images. For the following join example, we suppose that we have one feature vector for an image object. A similarity-based join performs for each image object of the left input table M_1 a similarity search in the right input table M_2. Let M_1 have the following two objects: $(x_1, (12, 15, 10))$ and $(x_2, (22, 32, 5))$ and M_2 has the objects $(y_1, (10, 14, 10))$ and $(y_2, (14, 16, 12))$. The similarity search is a range query with a range of 4 using the Euclidean distance. y_1, y_2 are the result objects of the similarity search of x_1 in M_2. For x_2 we do not have any similar objects in M_2. Thus, two result tuples are output: $(x_1, y_1, (12, 15, 10), (10, 14, 10))$ and $(x_1, y_2, (12, 15, 10), (14, 16, 12))$.

Cascading the joins allows one to merge more than two multimedia object sets (referred hereafter as multi join operation). Consider now a third image table M_3 with the following tuples $(z_1, (8, 15, 10))$ and $(z_2, (16, 20, 5))$. The result of the former join between M_1 and M_2 shall be joined to M_3 using a range query (range of 5 using the Manhattan distance). This subsequent join works as follows. First, we determine which feature vector of the join result M_1 with M_2 is used for the subsequent join, then we perform for each join result tuple a similarity search in M_3. We choose the feature vector of M_2 for the subsequent join. For the first tuple, $(x_1, y_1, (12, 15, 10), (10, 14, 10))$ we find z_1 to be similar, while z_2 is not similar enough. For the second tuple, $(x_1, y_2, (12, 15, 10), (14, 16, 12))$ we don't find any similar image object in M_3. Thus, $(x_1, y_1, z_1, (12, 15, 10), (10, 14, 10), (8, 15, 10))$ is the only result tuple of the multi join operation.

Let $M_1(p, fv_p, a_p)$ and $M_2(q, fv_q, a_q)$ be two *base* image tables, p, q are the image objects. $fv_p = f_1, f_2, ...$ is the set of feature vectors representing the low-level features of the object p (respectively for q). a_p and a_q are object types containing the attribute components that may be used to describe the objects. Formally, the **Similarity-based Image Join** is defined as:

(a) $join(M_1, M_2, f_p, f_q, \varepsilon) = X \Leftrightarrow X \in M_1 \times M_2 \wedge \forall((p, fv_p, a_p), (q, fv_q, a_q)) \in X \bullet q \in \varepsilon - similarity(M_2, p, f_p, f_q, \varepsilon)$.
(b) $join(M_1, M_2, f_p, f_q, k) = X \Leftrightarrow X \in M_1 \times M_2 \wedge \forall((p, fv_p, a_p), (q, fv_q, a_q)) \in X \bullet q \in$*k-NN-similarity*$(M_2, p, f_p, f_q, k)$.

$\varepsilon - similarity(M_2, p, f_p, f_q, \varepsilon)$ is the ε-similarity in M_2 computing all neighbors whose distance to the query image p is below a threshold ε. *k-NN-similarity*(M_2, p, f_p, f_q, k) is the *k-NN*-similarity computing the k-nearest neighbors to the query image p in M_2.

The result of a similarity-based join is a new intermediate image table M which contains the components of both joined image tables, thus the table schema expresses as: $M(p, q, fv_p, fv_q, a_p, a_q)$. The definition above can simply be extended to the join of one base table with an intermediate image table and to a join of two intermediate tables.

The **Multi Similarity-based Join** specifies $n \geq 1$ similarity-based joins. It is represented by a *linear processing tree* with the following syntax: $PT ::= M_i$ or $PT ::= join(PT_1, M_i, f_1, f_2, \varepsilon)$ or $PT ::= join(PT_1, M_i, f_1, f_2, k)$ where PT_1 is a linear processing tree.

For simplifying the notations and without loosing the generality of the optimization consideration, each image table M_i is used exactly once.

3 Related Work

This paper concentrates on methods for *efficient* processing of large-scale multi similarity-based join operations. To compute the similarity-based operations efficiently, the feature space is usually indexed using a multidimensional index data structure [6].

The similarity search in *large databases* may reveal time-consuming, if many index nodes have to be examined in a range search. To cope with this, recent works proposed a compact representation of the similarity join result [7]. While this limits the problem for a similarity-based join, it cannot solve the problem for multi join operations. If a nearest-neighbor search is employed, the computational complexity is in the many distance computations and in the size of the priority queues used for pruning. It becomes specially time- and space-critical for high-dimensional database spaces (see [8]). Recent works focused on improving the nearest-neighbor algorithm by using distance estimators, in order to reduce the storage requirements [9]. It has been proven that this can prune the priority queue without altering the output of the query. But the time complexity is still present. As a consequence, several authors considered the parallelization of similarity-based selection queries [10,11,12].

These works consider either *data distribution*, *space partitioning*, or *multiplexed index structures*. *Data distribution* assigns data rectangles/spheres to different computing nodes in a round robin manner or by a hash function. *Space partitioning* divides the space into partitions which are assigned to separate computing nodes. *Multiplexed index structures* are distributed over nodes with pointers across the nodes. Data distribution balances the load, while space partitioning activates few nodes. Multiplexed index structures are more flexible, they can balance the number of activated nodes vs load balancing among nodes. These strategies are well-researched. Coming to the processing of similarity-based join operations, Kosch et al. [3] considered simultaneously processing of two index structures, and Yu et al. [4] focused on dimension reductions and distance index structures. Both work did not consider parallelization strategies. In spatial databases, parallelization of joins have been considered by several authors (see Section 5 of Jacox et al. [13]). These works consider mainly the parallelization of the filter step of a spatial join which is special to spatial databases and cannot be directly transferred to multimedia databases. A recent work concentrated on the optimal placement of similarity-based multimedia operations in complex multimedia queries, a parallelization strategy is however only considered for the selection operators [14]. Distributed processing in IR systems considered also data distributions. Content replications and distributions for cluster-based architectures is for instance considered by Klampanos et al. [15].

Extending the parallelization of similarity-based selections to multi similarity-based joins is a challenging task. First, the index look-ups of the left input table image objects to the right image table must be parallelized. Second, the processing of the different join levels must be also parallelized in order to fully exploit the computing power of a parallel machine.

4 Large-Scale Processing

We want to effectively process large-scale *multimedia queries* specifying $n \geq 1$ similarity-based joins. For each $M_i(o_i, fv_i, a_i)$ $(1 \leq i \leq n)$ with $fv_i = f_{i1}, f_{i2}, ...$ being the set of feature vectors representing the low-level features of the object o_i, we assume that multidimensional index structures are available for each different $f \in fv_i$ to efficiently carry out the ε or k-*NN*-similarity. These multidimensional index structures shall be able to hold a large number of data points in possibly high-dimensional data spaces. From related work [6], *Data Partitioning* index structures are good candidates. We concentrate in this work on hierarchical index structures (e.g., X-,SS-, SR- and TV-trees), as they are the mostly used and tuned [16] index structures in multimedia database products.

Nested-Loop Index Join. The *method* for performing a similarity-based join is to apply the ε or k-*NN*-similarity for each object of the left input table M_1 as a query object o looking for its similar objects from the right input table M_2. The latter operation is implemented as an index look-up in the index structure of the right input table M_2.

The research problem we pose is how to process effectively large-scale operations. Our two main methods are: **Data Parallelism** of a single similarity-based operation and and **Pipeline Parallelism** of multiple similarity-based operations.

Data Parallelism. We suppose that we are disposing a cluster system with a high-speed interconnected computing nodes in a distributed memory architecture. Each node has b cores which access the shared memory on the node. We assume a shared disk architecture.

The processing is done in two phases. In the **index build phase** the feature vectors of all available image tables are distributed in round robin manner over the computing nodes. At each node, a local index structure is built. Thus, each distribution is complete and not overlapping. We have chosen the data distribution over the space partitioning parallelization (see Section 3), because it is load balancing for multi join processing. In a space partitioning approach, each level of the multi join processing introduces a load imbalance which could easily sum up to an important overall imbalance and thus leading to much higher-response times.

The processing of the similarity-based join is done in a master-worker manner (**processing phase**). The multimedia database server acts as master. It forwards the common feature vector for each object of the left input table M_1 to all computing nodes (workers), where an index look-up in the common feature vector index structure of the right input table M_2 is performed. The processing of each local join (on each node) can be done independently from the others. This scales very well. The results are sent back to the master. It prepares then for the subsequent join.

Example. Consider two image tables, M_1, M_2 and $a = 4$ computing nodes. The feature vectors of M_1 and M_2 are:

$fv_1 = \{dominant\_color, color\_histogram\}$ and
$fv_2 = \{dominant\_color, edge\_histogram\}$.

In the *index build phase*, the four available feature vectors (2 for M_1 and 2 for M_2) are distributed uniformly over the 4 nodes, thus each node holds 1/4 of the complete index structures. In the *processing phase*, we like to perform a *3-NN* similarity-based join of the two image tables. The common feature vector for M_1 and M_2 shall be the *dominant_color*. The master initiates the join by scanning M_1. For each tuple, it extracts the *dominant_color* feature vector and forwards the query information (vector and *3-NN*) to all nodes, where an index look-up on the local *dominant_color* index of the M_2 table is performed. The result (the tuple id) is returned to the master.

Pipeline Parallelism. We propose to process multiple similarity-based joins in pipelined fashion. The principle idea is taken from right-deep processing of join operations in parallel relational databases. The way of processing traditional joins works with exact matching, while similarity-based join processing works on possibly multiple feature vectors based on similarity matching. Thus, the distributed implementation of the pipelined similarity-based join cannot directly use right-deep processing implementations. We therefore originally designed a pipelined processing strategy for similarity-based joins with a fully threaded realization exploiting multi-core parallelism on different index structures (and feature vectors). The pipeline parallelism works as follows, once the result of a join arrives at the master node, the feature vector for the subsequent join is looked up the corresponding base table. It is immediately send to all nodes to probe against the next right input operator. Thus, the former join and the subsequent join execute on each node in parallel. In order to scale, the processing of the former join and the subsequent one is done by different threads allocated to different cores on each node. For instance, if each node has $b = 4$ cores, one may execute 4 join levels in parallel. The access to the shared memory could be the limiting factor for this parallelization. But, as the joins are performed on different right image tables, the access are not concurrently to the same data. This strategy scales well.

5 Efficient Parallel Implementation

The implementation of the multidimensional index structure and the similarity-based join operation is done in two main parts, the **Database Extension (SBMJLibrary)** and the **External Index Structure (GiSTServer)**.

Database Extension (SBMJLibrary): An Oracle 11g Database with the *Oracle Data Cartridge Interface Technology (ODCI)* offering extensibility interfaces is used. We can extend the query processing, type system and data indexing by calling external C,C++ or Java routines. We implemented the similarity-based

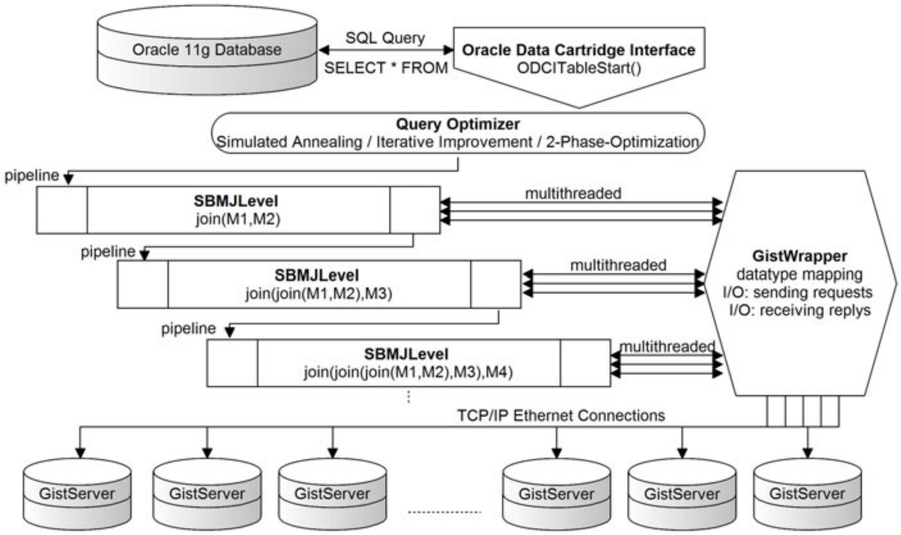

Fig. 1. Framework for the Similarity-based Join Execution

join operation as an external table function in C++. Figure 1 shows the framework for the similarity-based join execution. In particular one sees the components involved, these are the Oracle 11g Database, the ODCI Interface, the Query Optimizer, the SBMJLevel and the GiST-Framework. The ODCI Interfaces include external routines that are combined to a shared library. Each subsequent level of a similarity-based join is represented by an instance of a SBMJLevel. In each SBMJLevel, the specific join parameters (e.g., involved table indices, join type, etc.) and a reference to the GiSTWrapper are saved. New intermediate result tuples are computed by requesting index scans via GiSTWrapper calls. The GiSTWrapper is a singleton Object, designed to communicate with the GiSTServers.

Now, let's take a look on our realization. The join processing starts with the **Query Optimization**. Its task is to find a good ordering of the join operations, where few intermediate join result tuples are produced. In a previous paper [5], we introduced a cost model to compute the number of intermediate result tuples. This paper uses the cost model and implements an Iterative Improvement search strategy to compute a good ordering. The Iterative Improvement algorithm makes a specified number of local optimization. The algorithm starts at a random state and improves the solution by repeatedly performing swap- or 3-cycle-transformations until a local minimum or the maximal number of iterations is reached. The query optimization is the first part of the similarity-based join processing.

The **execution of the joins**, thus generating the result tuples is done in a *parallel and multithreaded manner* (**processing phase**). Each level of a similarity-based join (SBMJLevel) runs in a separate thread and is connected to an own input queue. An input queue contains intermediate result tuples from the upper level and acts as a data pipeline (*pipeline parallelism*). The input queue of the top level is initialized with feature vector / row-id pairs of the left input table of the deepest join. With each feature vector in the input queue, a new index scan request to the GiSTWrapper is started. The result of this query is a set of matching pairs, which are merged to a new intermediate result tuple, that is directly piped out to the input queue of the lower level. Considering that one level will always produce new tuples faster (or slower) than another, the level processing is also multithreaded to avoid a bottleneck. If there are more than one intermediate tuple in the input queue, the level runs a predefined number of worker threads for parallel index scan requests.

Now let us concentrate on the *GiSTWrapper*. GiST stands for *Generalized Search Tree* and is an extensible data structure framework, developed at the University of Berkeley [17]. The GiST-Library (libgist2.0) contains implementations for R-,R*,SS-,SR-,B- and NP-Trees and is designed for storing multidimensional index structures. Our extension, the GiSTWrapper is an interface between the database and the external index structure, which is based on the GiST-framework. The two main tasks of the GiSTWrapper are first to map the Oracle ODCI-datatypes to native C-datatypes and second to forward a query (e.g., index scan request) to the GiSTServers. Considering that the external index structure is uniformly distributed over multiple GiSTServers in the network, one query has to be sent to all GiSTServers by using a TCP/IP connection. For this purpose and to satisfy the criteria of a parallel and multithreaded execution, the GiSTWrapper runs a new worker thread for each connection. This thread holds the connection until the GiSTServer completes the computing of the query. The result of all threads is merged into a set of tuples, which are returned to the level thread. Note that in a *k-NN*-query, the result set contains *k * #GiSTServers* tuples and must be filtered before it is returned.

External Index Structure (GiSTServer). A GiSTServer communicates over our multithreaded **TCP/IP-Connection-Pool** which runs a configurable number of threads at start-up. When a new request arrives and a free thread is available, the execution of the request is allocated to that thread. If not, the request has to wait until a new thread is available. After a new request is received, it is forwarded to a **request handler**. The request handler parses each component, checks for errors and classifies the request into data manipulating and data querying requests. This determination is important for thread synchronization. In order to avoid that the threads must be fully synchronized when calling the GiST-framework, we preload multiple GiST-framework instances into the memory at start-up. This allows truly parallel query execution of threads on each node. This start-up handling is done by the **GiSTHolder-Pool** and the **GiSTHolder**. A GistHolder holds the entire GiST-framework in protected memory dynamically. Thus, a GiSTHolder can process a request on his own, in an independent memory area. The GiSTHolder-Pool is responsible for the scheduling of the GiSTHolder. In this manner, multiple data querying requests can be executed in parallel. On the completion of a request, either a status code (data manipulating) or the result tuples (data querying) are returned to the database.

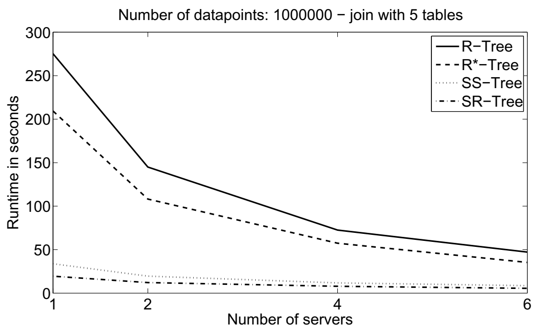

Fig. 2. Similarity-based Join Response Times with Real Data (1 Million), up to 6 Processors

As the TCP/IP Connection-Pool only supports synchronous communication, the reply is sent with the same connection as the request was received. Furthermore, we added several updates and bug fixes to the original GiST-framework, e.g., large file support on Linux platforms, so the GiSTServer can handle larger files than 2GB both on 64bit and 32bit compilates.

6 Experimental Results

We compiled and installed our implementation on a network cluster system. Our cluster system consists of one master node and 16 worker nodes, interconnected by a reliable Gigabit Ethernet network. To store the external index structure, we used a shared disk with 2TB disk space, accessible by each node. All nodes have the following hardware specification:

```
Operating System: openSUSE 10.3 (X86-64)
Kernel: 2.6.22.19-0.4-default x86_64
CPU(s): 1 Quad Core Intel Xeon E5405s
CPU Clock: 2000.069 MHz (each core)
Memory: 16078.3MB
```

We installed the Oracle 11g Database on the master node, the GiSTServers on the worker nodes.

For the following series of tests, we used two different data sets: first, *randomly generated data* (up to 10 millions) and second, *real feature vectors*, extracted from images of the Flickr Database (up to 1 million). The main purpose of the tests is to measure the response time improvements of the *processing phase* while increasing the number of computing GiSTServers.

We executed similarity-based joins with varying image data sets (random data, real data), four index structure types (R-Tree, R*-Tree, SS-Tree, SR-Tree), different table sizes (10^3 to 10^7) and join depths (1-4 cascading joins). We first found out that on every node, at least 10000 datapoints must be inserted to achieve an improvement. From that, the greater the table size the clearer the response times enhancements. In general, the greater the join depth, the better

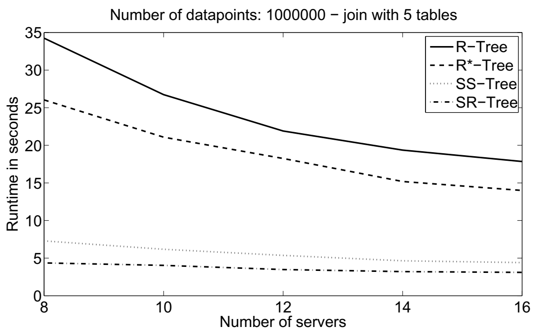

Fig. 3. Similarity-based Join Response Times with Real Data (1 Million), from 8 Processors

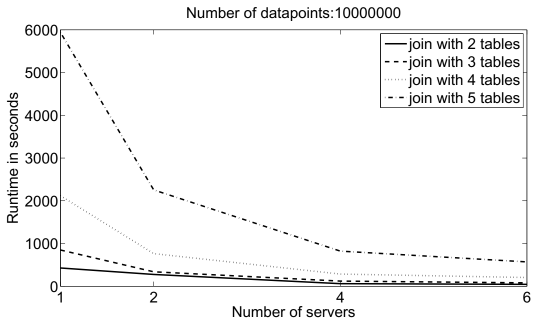

Fig. 4. Similarity-based Join Response Times with Random Data (10 Millions), up to 6 Processors

the response time enhancements. The response time highly depends on which tree family is used: Sphere-based trees deliver (compared to rectangle-based trees) the best performance. In this paper, due to the page size limit, the figures will be shown for the maximal amount of treated image objects: 10 millions in the case of randomly generated data and 1 million in the case of real data and for the maximal join depths of four.

The result of a cascading similarity-based join with 4 levels on the real data is shown in figures 2 and 3: $join(join(join(join(M_1, M_2), M_3), M_4), M_5)$. The feature vector in the real data is a 64-dimensional MPEG-7 *Scalable Color*. In order to regularize the number of result tuples, we set the size of the left input table of the deepest join M_1 to 10 datapoints. Each look-up table contained 1 million datapoints. We used a *1-NN* similarity search on the deepest level and a *3-NN* similarity search on the other levels. With this setup, 810 result tuples are produced. The Database Extension and the GiSTServers were configured to exploit the entire hardware capacity. Each of the 4 SBMJLevel were running 4 worker threads, so $16 * \#nodes$ threads were requesting the external index structures in parallel. Regarding the memory limitation on one node, the GiSTServers' thread-pool was set to 8 threads. The scaling is very good.

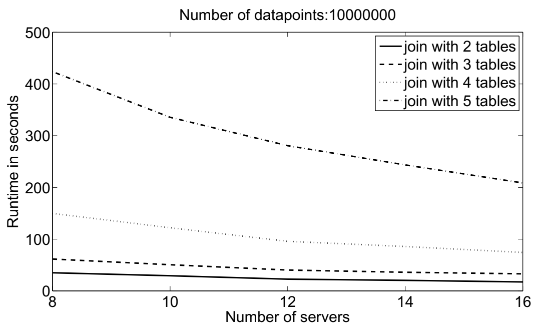

Fig. 5. Similarity-based Join Response Times with Random Data (10 Millions), from 8 Processors

Doubling the number of GiSTServers leads to just above *halving the response time*. The slightly lower values on a larger number of computing nodes results from a communication overhead and the bottleneck of the shared disk access.

Figures 4 and 5 show the results for the similarity-based joins on 5 randomly generated multimedia tables with 10 millions datapoints on an R*-Tree based index structure. We used the same thread- and similarity search configuration as in the previous tests. The response time improvements are similar to figures 2 and 3, except for nodes=1,2. We had to reduce the thread-pool of the nodes to 1 thread on 1 node and 3 threads on 2 nodes. This was necessary to avoid memory overflow.

In general, we observed that the running times for the real data sets are appreciable faster, which is due to the higher density areas in the real data sets. The difference decreases with a higher number of processors involved. Our parallelization strategy clearly scales with the size of the data sets joined and the number of processors used. It brings down the response times of complex queries to reasonable waiting periods with 10 millions data points involved.

7 Conclusion

This paper presented efficient parallelization methods for processing large-scale multimedia database operations. In special, similarity-based image join operations were efficiently been carried out by using data and pipeline parallelization strategies. In future works, we will extend the parallelization framework of GiST with further index structures, e.g., from the Windsurf framework[1]. We also intend to integrate so called "distance joins" between two image input sets, e.g., to compute the K-closest pairs of the two image input sets, ordered by the distance of objects in each pair.

[1] http://www-db.deis.unibo.it/Windsurf/

References

1. Lew, M.S., Sebe, N., Djerba, C., Jain, R.: Content-based multimedia information retrieval: State-of-the-art and challenges. ACM Transactions on Multimedia Computing, Communications, and Applications 2(1), 1–19 (2006)
2. Datta, R., Joshi, D., Li, J., Wang, J.Z.: Image retrieval: Ideas, influences, and trends of the new age. ACM Computing Surveys 40(2), 1–60 (2008)
3. Kosch, H., Atnafu, S.: Processing a multimedia join through the method of nearest neighbor search. Inf. Process. Lett. 82(5), 269–276 (2002)
4. Yu, C., Cui, B., Wang, S., Su, J.: Efficient index-based knn join processing for high-dimensional data. Information and Software Technology 49, 332–344 (2007)
5. Kosch, H.: Optimizing similarity-based image joins in a multimedia database. In: Proceedings of the ACM International Workshop on Very-Large-Scale Multimedia Corpus, Mining and Retrieval, VLS-MCMR 2010, pp. 37–42. ACM (2010)
6. Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann (2006)
7. Bryan, B., Eberhardt, F., Faloutsos, C.: Compact similarity joins. In: Proceedings of the IEEE International Conference on Data Engineering, ICDE 2008, pp. 346–355. IEEE (2008)
8. Samet, H.: Techniques for similarity searching in multimedia databases. PVLDB 3(2), 1649–1650 (2010)
9. Bustos, B., Navarro, G.: Improving the space cost of k-NN search in metric spaces by using distance estimators. Multimedia Tools and Appl. 41(2), 215–233 (2009)
10. Berchtold, S., Böhm, C., Braunmüller, B., Keim, D.A., Kriegel, H.-P.: Fast parallel similarity search in multimedia databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 1–12. ACM (1997)
11. Alpkocak, A., Danisman, T., Ulker, T.: A Parallel Similarity Search in High Dimensional Metric Space Using M-Tree. In: Grigoras, D., Nicolau, A., Toursel, B., Folliot, B. (eds.) IWCC 2001. LNCS, vol. 2326, pp. 166–171. Springer, Heidelberg (2002)
12. Manjarrez-Sanchez, J., Martinez, J., Valduriez, P.: Efficient Processing of Nearest Neighbor Queries in Parallel Multimedia Databases. In: Bhowmick, S.S., Küng, J., Wagner, R. (eds.) DEXA 2008. LNCS, vol. 5181, pp. 326–339. Springer, Heidelberg (2008)
13. Jacox, E.H., Samet, H.: Spatial join techniques. ACM Transactions on Database Systems 32(1) (2007)
14. Wu, Z., Cao, Z., Wang, Y.: Multimedia selection operation placement. Multimedia Tools and Appl. 54(1), 69–96 (2011)
15. Klampanos, I.A., Jose, J.M.: An Evaluation of a Cluster-Based Architecture for Peer-to-Peer Information Retrieval. In: Wagner, R., Revell, N., Pernul, G. (eds.) DEXA 2007. LNCS, vol. 4653, pp. 380–391. Springer, Heidelberg (2007)
16. Shen, H.T., Huang, Z., Cao, J., Zhou, X.: High-dimensional indexing with oriented cluster representation for multimedia databases. In: Proceedings of the IEEE International Conference on Multimedia and Expo, ICME 2009, pp. 1628–1631. IEEE (2009)
17. Hellerstein, J.M.: Generalized Search Tree. In: Encyclopedia of Database Systems, pp. 1222–1224. Springer, US (2009)