

Joining Ranked Inputs in Practice*

Ihab F. Ilyas

Walid G. Aref

Ahmed K. Elmagarmid

Department of Computer Sciences, Purdue University
1398 Computer Science Building
West Lafayette IN 47907-1398
USA
{ilyas,aref}@cs.purdue.edu

Hewlett Packard
1501 Page Mill Rd.
Palo Alto, CA 94304
USA
ahmed_elmagarmid@hp.com

Abstract

Joining ranked inputs is an essential requirement for many database applications, such as ranking search results from multiple search engines and answering multi-feature queries for multimedia retrieval systems. We introduce a new practical binary pipelined query operator, termed NRA-RJ, that produces a global rank from input ranked streams based on a score function. The output of NRA-RJ can serve as a valid input to other NRA-RJ operators in the query pipeline. Hence, the NRA-RJ operator can support a hierarchy of join operations and can be easily integrated in query processing engines of commercial database systems. The NRA-RJ operator bridges Fagin's optimal aggregation algorithm into a practical implementation and contains several optimizations that address performance issues. We compare the performance of NRA-RJ against recent rank join algorithms. Experimental results demonstrate the performance trade-offs among these algorithms. The experimental results are based on an empirical study applied to a medical video application on top of a prototype database system for streaming multimedia data. The study reveals important design options and shows that the NRA-RJ operator outperforms other pipelined rank

join operators when the join condition is an equi-join on key attributes.

1 Introduction

A growing number of database applications require the processing of ranking queries based on multiple attributes. In the context of multimedia retrieval, predicates often involve image similarity matching with respect to several features. Users may present an example image, and query the database for images "most similar" to the example based on color and texture. Although each database image object can easily be ranked for color and texture separately, results must be presented to the user in a combined similarity order. Another example from information retrieval is the search for documents containing search topics from multiple sources [19]. While each source provides retrieved documents sorted by relevance, the collection of retrieved documents returned to the user must be sorted in a combined relevance order. Other examples include information retrieval based on multiple keywords, CAD similarity searches, geometrical databases, medical imaging and molecular biology.

The query evaluation model used for a similarity search does not generally return a collection of exact matches, but rather a ranked collection of results with a score attached to each result. Users are usually interested in only the top-ranked results. The aggregate score for a given result is obtained by combining the scores of several *atomic* similarity rankings, where the atomic ranking is based on a single feature or attribute of the database object. For example, atomic rankings are produced in single-feature similarity queries in multimedia retrieval [5, 8, 11] and in document ranking based on a single search topic in information retrieval applications. The challenge for similarity search is the ranking of results based on an overall aggregate score, using several atomic rankings as input. A naïve approach is to calculate a global score for each object in the database and sort the objects according to the computed score. This approach is not scalable, how-

The support of the National Science Foundation under Grants IIS-0093116, EIA-9972883, IIS-9974255, and EIA-9983249 and the Department of Navy under Grant N00164-00-C-004 are gratefully acknowledged.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

ever, and the performance of the system deteriorates rapidly as the size of the database increases.

Many algorithms have been proposed in the literature to address aggregation ranking, including Fagin’s algorithm [6], the TA, CA and NRA algorithms [7], the Quick-Combine algorithm [9], the multi-step aggregation algorithm [15] and the Stream-Combine algorithm [10]. Some prototype systems have incorporated these algorithms to answer aggregated rank queries, such as the IBM GARLIC middleware [18], which uses Fagin’s algorithm. A simulation of Fagin’s algorithm is used in [4] as a “filter condition” for querying multimedia repositories; the Quick-Combine and multi-step algorithms are used in multimedia retrieval for answering multi-feature queries [9, 15]; and the Stream-Combine algorithm is used in middleware for heterogeneous environments [10]. Recently, Natsev et al. introduced the J^* algorithm, an incremental algorithm to join ranked inputs based on the A^* search algorithm. In the rest of this paper we refer to these algorithms as *rank-join* algorithms.

Implementing the rank-join algorithms in database systems extends the database ability to answer a wider range of user queries needed by many recent applications. Two alternatives exist for implementing these algorithms in a database system: using *table functions* or encapsulating the algorithm in a physical query operator. With the first approach, rank-join algorithms are implemented in the *application* level (i.e., outside the SQL engine) and SQL table functions [16] are examples of this type of implementation. Since there is no straightforward method for pushing query predicates into table functions [17], the performance of this query is severely limited and the approach does not give enough flexibility in optimizing the issued queries. The second approach for implementing a rank-join algorithm in a database system is to define a query *operator* that can be part of the query execution plans. The operator will encapsulate the rank-join algorithm in its *GetNext* operation; each call to *GetNext* should report the next top element from the input ranked inputs. As we will show in this study, only *incremental* and *pipelined* aggregation algorithms can be of practical use in query evaluation pipelines.

Our Contribution

- We propose a new practical pipelined query operator NRA-RJ. We made necessary modifications to an efficient algorithm for joining ranked inputs that assumes no random access is available on input streams. The new operator will help incorporating this type of join in ordinary query plans and hence can be adopted by real database engines.
- We compared between the state-of-art algorithms in joining ranked inputs that can be realized as

query operators. The study compares the time and the space complexity of these algorithms as well as different optimizations that help in enhancing the overall performance. Different trade-offs are discussed.

- We conducted an empirical study for these algorithms on real data in the context of multimedia retrieval. The experiments explored the scalability of such algorithms and different performance metrics.

Organization of the Paper The rest of the paper is organized as follows. Section 2 gives alternatives for implementing the rank-join algorithms in a database system and discusses the advantages of building a physical query operator to realize a rank-join algorithm. Section 2 also gives an overview on two efficient rank-join algorithms namely, the NRA algorithm [7] and the J^* algorithm [14]. In Section 3 we introduce a physical pipelined binary query operator, NRA-RJ, for rank join based on an adaptation of the NRA algorithm. Section 4 presents an optimization heuristic to the basic NRA-RJ algorithm to enhance its performance and scalability to long query pipelines. A comparison between the J^* and NRA-RJ is described in Section 5. The two operators and several optimizations are evaluated through an empirical study in Section 6. Section 7 contains a summary of our findings, recommendations, and concluding remarks.

2 Rank-Join Algorithms as Query Operators

Implementing a rank-join algorithm as a binary pipelined query operator is very appealing for query optimization and allows for handling nested joins and views efficiently. Moreover, the operator will permit greater flexibility in generating candidate execution plans as opposed to the use of table functions. The encapsulation of the rank-join algorithm into a sequence composed of multiple binary operators makes it possible to shuffle the evaluation plan operators in seeking the best plan. Moreover, it is possible to apply predicates on partial ranking results by pushing selection in different levels in the rank-join pipeline.

For a rank-join algorithm to be implemented as a pipelined query operator, the algorithm should support two key properties. First, the algorithm should be *incremental*. An incremental rank-join algorithm does not depend on specifying the number of required results beforehand, rather, it provides the next result whenever called for. The second property a rank-join algorithm should support is the *pipelining* property. For a query operator to be part of a pipeline, the output of an operator should be a valid input to the next operator in the pipeline. The pipelining property allows for realizing join hierarchies and nested views, and hence a wider range of query evaluation plans.

Table 1 summarizes the recent rank-join algorithm and compares their basic properties. First, Algorithm FA [6] was introduced by Fagin as an efficient solution to the problem. Algorithms TA [7] (Fagin et al.), Multi-step [15] (Nepal et al.) and Quick-Combine [9] (Güntzer et al.) are equivalent and are an enhancement over the FA algorithm. These four algorithms depend on the availability of random access to the ranked inputs and hence are not pipelined; random access is not possible when the input arrives as output from another execution of the algorithm in the query pipeline. They can only be executed on the leaf level of the query evaluation plan. For example, realizing the FA algorithm in the IBM GARLIC middleware [18] was limited to a query operator that can exist only on the leaf level of the query evaluation plan.

Algorithms NRA [7] (Fagin et al.) and Stream-Combine [10] (Güntzer et al.) do not require random access to the ranked inputs. While Stream-Combine can be realized easily as a pipelined operator, the NRA algorithm is not pipelined since the output does not have exact grades associated with the reported objects. Hence, the output of the NRA algorithm cannot serve as a valid input to another NRA execution. The algorithm NRA-RJ proposed in this paper modifies the NRA algorithm to allow for pipelining. The J^* algorithm [14] (Natsev et al.) also does not require random access to the input and can easily be realized as a pipelined query operator. One difference between the J^* algorithm and the other algorithms in the literature is the ability of J^* to handle general join conditions while other algorithms assume that the same set of objects are ranked differently in each input and hence are limited to equi-join on key attributes.

Since we focus only on this class of rank-join algorithms that do not require random access to their input streams, we will elaborate on the last three algorithms in Table 1. Algorithm Stream-Combine is very similar to the NRA algorithm except for the fact that it requires a reported object to be seen, through sorted access, in all input streams. The NRA algorithm does not require this condition, and hence has a faster termination condition as we will discuss shortly. This will limit the comparison to the NRA algorithm (modified to be realized as a query operator) and the J^* algorithm.

To the best of our knowledge, these are the only efforts to introduce the rank-join algorithm as a *pipelined* query operator. We will refer to the *binary* operator that encapsulates the NRA algorithm in [7] as the NRA-RJ operator and to the binary operator that encapsulates the J^* algorithm as the J^* operator. For the paper to be self-contained, we briefly present the NRA and the J^* algorithms. The reader is referred to [7, 14] for more details.

The NRA Algorithm The NRA algorithm views the database as m input lists where each list consists of

Algorithm	No Random Access	Pipelineable	Join Condition
FA	No	No	key
TA	No	No	key
Multi-step \approx TA	No	No	key
Quick-Combine \approx TA	No	No	key
Stream-Combine	Yes	Yes	key
NRA	Yes	No	key
J^*	Yes	Yes	General

Table 1: Rank-Join Algorithms

objects associated with grades and objects are sorted in descending order on these grades. Let t be a weighting function to compute the overall grade of an object by applying t to all the individual grades of this object. The NRA algorithm [7] (no-random-access) visits objects from the m input lists in parallel. At depth d (i.e., when the first d objects have been visited across all m streams), the *bottom* values $\underline{x}_1^{(d)}, \underline{x}_2^{(d)}, \dots, \underline{x}_m^{(d)}$ are maintained as the grades last seen from each input list. For an object R with l discovered fields $x_1, \dots, x_l, l \leq m$, we compute the worst grade as $W^{(d)}(R) = t(x_1, x_2, x_3, \dots, x_l, 0, \dots, 0)$ and the best grade as $B^{(d)}(R) = t(x_1, x_2, x_3, \dots, x_l, \underline{x}_{l+1}, \dots, \underline{x}_m)$. Objects encountered thus far are sorted in descending order according to their worst grade, where ties are broken using an object's best grade. If the top k objects are required, and we let $M_k^{(d)}$ be the k^{th} largest worst grade, then the algorithm halts when no object outside the top k objects encountered thus far has a best grade greater than $M_k^{(d)}$.

Using the NRA algorithm directly in the implementation of a pipelined query operator is complicated by two problems. First, the algorithm depends on a pre-defined value for k , the number of top results to be retrieved. What we need is an *incremental* version of the algorithm which produces the *next top* object when needed by the caller. The second problem is that the output from the algorithm does not have exact grades associated with the output objects. Instead, each object has a range from worst grade to best grade. This prevents pipelining the operator in the query plans, since the exact ranks (grades) will be available only from the source input streams.

The J^* Algorithm The J^* algorithm is introduced in [14] by providing the method *GetNext* that reports the next top join combination in each call. The algorithm is based on the A^* class of search algorithms. As in the A^* search algorithm, the cost of the path leading to the final answer is divided into two parts: the first part is the actual cost encountered thus far, and the second part is an estimate of the cost before reaching

```

J*: GetNext()
Given: a queue buffer Q
1. LOOP
2. IF Q is Empty
3.   RETURN Null
4.  head = Q.top.
5.  IF head is a complete state
6.   RETURN head
7.  head2 = a copy head.
8.  X = an unassigned variables in head2.
9.  IF no tuples available for this stream
10.  tuple = stream.GetNext.
11.  Assign tuple to X and compute score.
12.  IF the assignment is valid
13.   Push head2 in Q.
14.  Let X in head points to the next tuple in the
    corresponding stream.
15.  Push head into the Q
16. END LOOP

```

Table 2: The J^* GetNext operation.

the final answer. The J^* algorithm works as follows. For each input stream, a variable is defined whose set of possible values are the tuples from the corresponding stream. The goal is to find a valid assignment for all the variables that maximizes the total score, which corresponds to finding the top valid join combination. The term *state* is defined as a set of variable assignments, and a *complete state* is a state that instantiates all the variables. Otherwise, the state is called a *partial state*. To find the next top join combination, the algorithm maintains a priority queue of partial and complete join results ordered on *upper bound* estimates of the final combination scores. At each step, the state on top of the queue is processed in an attempt to assign one of the unassigned variables by pulling the next tuple from the corresponding input stream. The algorithm terminates when a complete state (a valid join combination) appears on top of the priority queue. Table 2 gives a layout of the J^* algorithm. The recursive call to *GetNext* in line 10, and the fact that the algorithm returns the answer along with the exact combined score, allows the algorithm to work well with join hierarchies.

3 The NRA-RJ Operator

The physical query operator, NRA-RJ (No-Random-Access Rank Join), is a pipelined binary operator that implements a modified version of the NRA algorithm in [7].

We propose an incremental, pipelined version of the NRA algorithm that can be used to implement the *GetNext* operation of the NRA-RJ operator. We present the modified algorithm in terms of the NRA-RJ operations *Open*, *GetNext*, and *Close*. The internal

state information needed by the operator consists of a *priority* queue which holds the objects encountered thus far. The objects are sorted on worst grade in descending order, and ties are broken using the best grade (and then arbitrarily for ties on the best grades). In order to allow for pipelining, inputs to the algorithm may be source streams or output streams from other algorithm executions. Therefore, each object in the input streams is associated with a range of grades from worst grade w to best grade b (where $w = b$ for exact grades). At depth d (d is the number of objects retrieved from each input stream), the proposed algorithm maintains the bottom values $b_1^{(d)}$ and $b_2^{(d)}$. The worst grade of an object R is computed as $t(w_1, w_2)$, where t is the weighting function and w_i is either the worst grade of the object according to input i , or 0 if the object has not yet been encountered in input stream i . Similarly, the best grade of an object R is computed as $t(b_1, b_2)$, where b_i is the best grade of the object according to input stream i , or $b_i^{(d)}$ if the object has not yet been encountered in input stream i .

In the *Open* operation, the operator initializes the internal state information and opens the left and right child iterators. The *Close* operation destroys the state information and closes the input iterators.

The algorithm for the *GetNext* operation is given in Table 3. *GetNext* is the core of the rank join operator.

For clarity, we list the differences between the *GetNext* algorithm and the original Non-Random-Access algorithm in [7]:

- In general, the *GetNext* algorithm works for any query evaluation plan including allowing multiple inputs and composite inners (*bushy tree*), where the grades of the input streams are expressed as ranges. For simplicity, we will consider only the case when the algorithm is a binary operation, i.e., the number of input streams is limited to two. We will also consider only query evaluation plans that are *left deep trees*, where The right input list is a *source* input stream, which provides the operator with the ranked objects and their *exact* grades. On the other hand, the left input stream may not be a source list, since it can be the output of another NRA-RJ operator. In this case, the grade is expressed as a *range* from worst to best grade. Hence, the *GetNext* algorithm must be able to handle a grade range instead of an exact grade from the left iterator. Note that this restriction is merely for practical reasons; the algorithm still holds for more than two inputs.
- The parameter k (the number of requested output objects) is not known in advance, rather it increases for each call to *GetNext*. The modified algorithm first checks if it can report another object from the priority queue without violating the stopping condition. Otherwise it has to move

deeper into the input streams to retrieve more objects.

NRA-RJ: GetNext()

Given: a weighting function t .
a queue buffer Q

1. Threshold = 0.
2. IF Q is not Empty.
3. tuple = Q .Top.
4. W = tuple.WorstGrade.
5. B_{max} = Maximum Best Grade in Q -tuple.
6. IF $W \geq \text{Max}(B_{max}, \text{Threshold})$
7. RETURN tuple.
8. LOOP.
9. leftTuple = Left.GetNext(depth).
10. rightTuple = Right.GetNext(depth).
11. leftBottom = leftTuple.BestGrade.
12. rightBottom = rightTuple.BestGrade.
13. Threshold = $t(\text{leftBottom}, \text{rightBottom})$.
14. Check if tuple were seen before.
15. IF tuples exist in Q
16. Update Worst grade with exact grade.
17. Reinsert tuple.
18. For each Object in the queue:
19. Update the BestGrade.
20. tuple = Q .Top.
21. W = tuple.WorstGrade.
22. B_{max} = Maximum Best Grade in Q -tuple.
23. IF $W \geq \text{Max}(B_{max}, \text{Threshold})$
24. BREAK LOOP.
25. ENDLLOOP
26. Remove tuple from top of Q
27. RETURN tuple.

Table 3: The NRA-RJ GetNext operation.

The algorithm in Table 3 begins by checking the buffer (priority queue) to see if an object can be reported. An object can be reported if its worst grade is still greater than the best grades of all other objects. The maximum best grade for objects encountered thus far is obtained from the buffer. For objects not yet encountered, a threshold value can be used as an upper bound of the maximum possible best grade. The threshold is obtained by applying the weighting function to the best grades of the last encountered left and right objects. The maximum best grade in the buffer is maintained so that a scan of the whole buffer is not necessary for each call. To deal with grade ranges, the algorithm uses the best grades from the input streams to update the bottom values and to update the best grade of objects in the buffer.

We introduce a heuristic to the NRA-RJ algorithm in Table 3 to reduce the unnecessary ranking overhead at the early stages of the pipeline. We refer to the problem of performing excessive ranking at the early stages of the query pipeline as the *local ranking* prob-

lem. We elaborate on the *local ranking* problem and the heuristic to solve it in the next section.

4 Optimizing the NRA-RJ Operator

The NRA-RJ as given in Table 3 suffers from a computational overhead as the number of pipeline stages increases. To understand this problem we elaborate on how NRA-RJ works in a pipeline of 3 inputs assuming three input streams, L_1 , L_2 and L_3 . When the top NRA-RJ operator, OP_1 , is called for the next top ranked object, several *GetNext* calls for the left and right children are invoked. According to the NRA-RJ algorithm described in Table 3, at each step, OP_1 gets the next tuple from its left and right children. Hence, OP_2 will be required to deliver as many top ranked objects of L_2 and L_3 as the number of objects retrieved by L_1 . These excessive calls to the ranking algorithm in OP_2 result in retrieving more objects from L_2 and L_3 than necessary and accordingly, result in larger queue sizes and more database accesses.

One solution is to unbalance the depth step in the operator children. We change the NRA-RJ *GetNext* algorithm to reduce the local ranking overhead by changing the way it retrieve tuples from its children; for each p tuples accessed from the right child one tuple is accessed from the left child. The idea is to have less expensive *GetNext* calls to the left child, which is also an NRA-RJ operator. Using different depths in the input streams does not violate the correctness of the algorithm [7], but will have a major effect on the performance. This optimization significantly enhances the performance of the NRA-RJ operator as will be demonstrated in Section 6.3. Through the rest of the paper we will call p the *balancing factor*. Choosing the right p is a design decision and depends on the data and the order of the input streams, but a good choice of p boosts the performance of NRA-RJ.

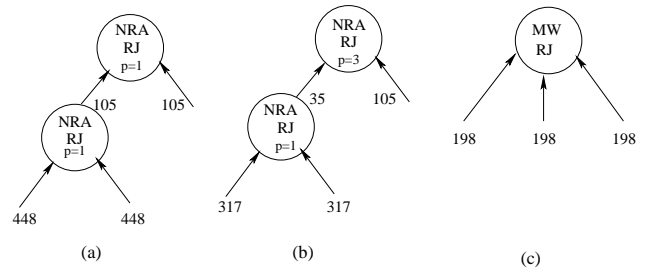


Figure 1: The effect of applying the heuristic to solve the local ranking problem in NRA-RJ.

For example, for a typical query with three ranked inputs, we compare between the total number of accessed tuples by the NRA-RJ operator before and after applying the heuristic. Also, as a reference, we compare the NRA-RJ operator with a direct implementation of the NRA Algorithm, the MW-RJ Oper-

ator. The MW-RJ operator is a multi-way rank-join operator. Figure 1 shows the number of retrieved tuples for each case. In the plan in Figure 1 (a), p is set to 1 for both NRA-RJ operators. According to a real data example execution of this query pipeline, the top NRA-RJ operator retrieves 105 tuples from both children, hence the top 105 tuples are requested from the NRA-RJ child operator, which has to retrieve 448 tuples from each of its children, for a total of 1001 tuples. In the plan in Figure 1 (b), p is set to 3 for the top NRA-RJ operator. While retrieving the same answers, the total number of tuples retrieved is 739 tuples, which is much less than that of the NRA-RJ before applying the heuristic since the top NRA-RJ operator requested only 35 tuples from its left child. The plan in Figure 1 (c) shows the number of tuples retrieved by the MW-RJ operator, which requires 198 tuples from each of its three children for a total of 594 tuples.

5 NRA-RJ vs. J^*

Both the J^* and the NRA-RJ operators implement a joining algorithm that joins multiple ranked inputs. The two operators are binary and pipelined and can be integrated easily in query evaluation plans. On the other hand, the two operators were designed for different problem settings. The NRA-RJ operator requires the existence of a *key* as the join attribute. More precisely, the NRA-RJ operator joins two streams of the same objects ordered differently. In contrast, the J^* operator can actually join different objects under arbitrary joining conditions, and hence can be used in a wider range of applications. Since it is more general, the J^* operator suffers from larger space requirements in the worst case, as we will show in the next sections. Both algorithms have proven to be *instance-optimal* with respect to database access cost, where instance optimality is a stronger optimality condition defined by Fagin et al. [7].

In this section we highlight important design differences between the NRA-RJ operator and the J^* operator. Two important design aspects of an operator are the stopping criteria and the space requirements. The stopping condition has a direct effect on the number of database accesses made by the operator. The goal is to stop as soon as we have enough information to report the next top-ranked object. The space requirement is an important design parameter in a database engine, since the maximum space required by an operator is translated into the quantity of resources that must be allocated. We conduct a worst case analysis for both the NRA-RJ and the J^* operators. For completeness, we also provide a best case analysis on the buffer size of both operators. The following comparisons are made for the problem of joining multiple sets of the same objects ordered differently in each ranked input. For arbitrary join conditions the J^* operator

becomes the only choice.

5.1 Stopping Condition

The idea behind the rank-join algorithms is to stop as early as possible, without the need to actually sort all the streams or visit more objects than needed. The algorithm implemented by the NRA-RJ operator achieves just that by introducing the *Worst Grade* and the *Best Grade* of an object. The idea is to stop when we are guaranteed that this object cannot have less overall score than any other object in the database, even if not *all* streams have been seen so far. In contrast, the J^* operator requires that an object must be seen in every input stream before reporting it. This constraint can be seen from the algorithm in Table 2; only *complete* states on top of the queue can be reported. To illustrate the early stopping criteria of the NRA-RJ we give the following example. Given two ranked inputs $L_1 = (R_1 : 10, R_2 : 5, R_3 : 4, R_4 : 3)$ and $L_2 = (R_2 : 5, R_3 : 4, R_4 : 3, R_1 : 1)$, where each object is attached to a different score in each list (the scores of R_1 are 10 in L_1 and 1 in L_2). Let W_i and B_i denote the worst grade and the best grade of object R_i , respectively. We use a simple monotone function $t(a, b) = a + b$ to calculate the overall score, and after two steps by the NRA-RJ operator, the best grade and the worst grade of objects seen so far are as follows: $W_1 = 10, B_1 = 14$; $W_2 = 10, B_2 = 10$; and $W_3 = 4, B_3 = 9$. According to the stopping criteria of the NRA-RJ operator, both R_1 and R_2 can be reported as the first top objects. Note that object R_1 has not been encountered yet in the input L_2 but it is guaranteed to have a worst grade that is larger than the best grade of any other object. For the same example using the J^* operator, R_1 cannot be reported as an output before accessing all objects in the input list L_2 .

5.2 Space Complexity

Rank-join algorithms with no random access suffer from the problem of unbounded buffer requirements for tracking the best grade of the objects encountered so far. Thus, the operator may require bookkeeping tasks for a huge queue containing these objects before it can report the next object. When comparing the space requirements of the two operators, a worst case analysis is used to estimate the maximum size of the buffer that should be reserved in order to correctly report the output objects. Due to the ability of the J^* operator to handle general join conditions, it has to consider more join combinations in the maintained priority queue. Hence, the space required by the J^* operator is larger than that required by the NRA-RJ, as shown in the following subsections and in the performance evaluation in Section 6.

5.2.1 Worst Case Analysis

In the worst case, a rank-join algorithm cannot report any object unless *all* objects from the input ranked lists have been seen. Let L_1 and L_2 be the two source rank lists for objects $\{R_1, R_2, \dots, R_n\}$. For simplicity, let the grade of an object in a list be $n+1-\text{rank}$, let $L_1 = (R_1, R_2, R_3, \dots, R_n)$ and let $L_2 = (R_n, R_{n-1}, R_{n-2}, \dots, R_1)$. The grades of object R_1 in lists L_1 and L_2 are n and 1 , respectively. Let the weighting function $t(a, b) = a + b$, i.e., a simple monotone function.

The NRA-RJ Operator: Assume we have moved to depth d in the two lists, and that the objects encountered so far from lists L_1 and L_2 are (R_1, R_2, \dots, R_d) and $(R_n, R_{n-1}, \dots, R_{n-d+1})$, respectively. Our goal is to report the top-most object. The maximum worst grade value encountered so far is the worst grade of object R_1 , computed as $W_1 = t(n, 0) = n$. Hence, R_1 is on top of the queue and we can report it only if the maximum best grade for all other objects is less than W_1 . The maximum best grade for objects encountered so far (other than R_1) is that of object R_2 , computed as $B_2 = t(n-1, n-d+1) = 2n-d$. According to the stopping criteria of NRA, we can stop only when $2n-d < n$, and that occurs at depth $d = n$, i.e., we must move entirely through both lists with a buffer size of n objects.

The J^* Operator: As discussed earlier in this section, the J^* operator solves a more general problem than the NRA-RJ, where it can handle arbitrary join conditions. To be able to compare both operators, we will consider only the case when the join condition is on a key attribute (for example, self-join). In order to see the space complexity of the operator in dealing with the input lists L_1 and L_2 , refer to Figure 2. In the J^* algorithm with two input lists, a state can be either *complete*, incomplete with one unassigned variable (we will refer to this state as *half-complete*), or incomplete with two unassigned variables (we will refer to this state as *incomplete*). When processing a half-complete state, two states are produced. The first state is a complete state, which is inserted only if it is a valid join combination (when the two variables represent the same object in the case of self-join). The second state is another half-complete state and it is inserted in the queue. In Figure 2, triangles represent half-complete states, while circles represent incomplete states. Processing an incomplete state produces two states, a half-complete state and another incomplete state, and both of them are kept in the queue. In the example, when L_1 and L_2 are the two inputs, it is easy to see from Figure 2(a) that all valid half-complete states must be present in the queue before reporting any objects (all objects have the same global score). When processing these half-complete states, each state will produce a valid complete state that will be kept

in the queue in addition to another half-complete state that is also inserted in the queue, yielding a buffer size of $2n-1$ states. Given that each state holds two tuples, the total buffer size is $4n-2$ tuples, which is larger than that of the NRA-RJ operator.

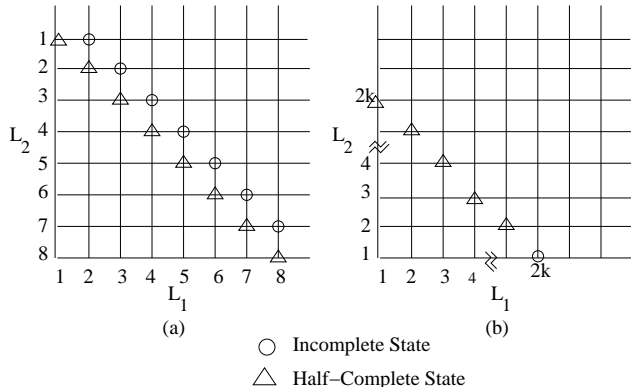


Figure 2: Space complexity of the J^* operator.

5.2.2 Best Case Analysis

For the best case analysis, we compare the two operators when the two ranked inputs are identical. The NRA-RJ does not need to keep any reported tuples therefore, the buffer size is always *zero*. For the J^* algorithm, the maximum buffer size is twice the size of the required results. To see that, we refer to the previous example with $L_1 = L_2 = (R_1, R_2, \dots, R_d)$. Figure 2(b) shows the type of the states that can be stored in the buffer at the time the object R_k can be reported. Because the J^* algorithm has to keep all possible combinations in the buffer, the buffer will have at least $2k$ states before reporting the k^{th} object.

6 Empirical Results

In Section 5 we highlighted several design differences between the NRA-RJ and J^* operators, and their impact on performance. In this section we describe several evaluation experiments in comparing the two binary pipelined operators, NRA-RJ and J^* , using real world data.

The experiments are based on our research platform for a complete video database management system running on a Sun Enterprise 450 with 4 UltraSparc-II processors running SunOS 5.6 operating system. The research platform is based on PREDATOR [20], the object relational database system from Cornell University. Shore [1] is the underlying storage manager, where the digital video is stored as a large object and is defined as an abstract data type (ADT). Video visual features are stored as high-dimensional *vectors* that must be indexed using a high-dimensional indexing scheme. To accommodate the high-dimensional indexing, we extended the indexing capabilities of Shore by

adding the GiST general indexing framework [12]. We used the GiST implementation of the SR-tree [13] as the indexing technique. The nearest-neighbor search operator is implemented as an incremental NN search query on the SR-tree.

The following experiments are conducted on the database table *Features*, which contains 100,000 records of features extracted from video frames. The feature fields include color histogram in YUV format (a vector of 32 dimensions), texture tamura (a vector of 16 dimensions) and texture edges (a vector of 9 dimensions). We use the query evaluation plan, given in Figure 3, to evaluate the proposed operators. The plan has m NN operators on m different visual features. $m-1$ rank-join binary operators are used, where the results of one operator are pipelined to the next operator in the pipeline.

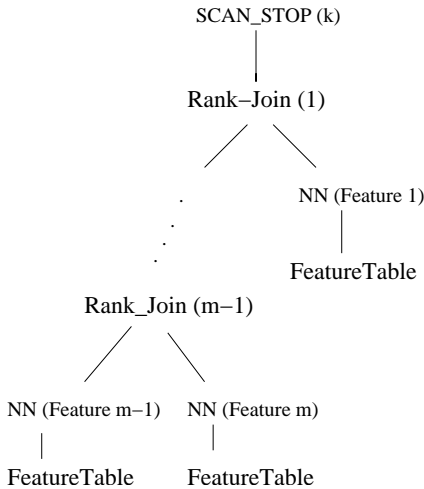


Figure 3: The query plan used in the experiments.

We use the following query to evaluate the performance of the three rank-join operators:

Q: Retrieve the k most similar video shots to a given image based on m visual features.

where m varies from 2 to 6 features and k varies from 5 to 100. Note that the number of requested results, k is not an input to the rank-join operator. We limit the number of reported answers to k by applying the *Stop-After* query operator introduced by Carey and Kossmann [2, 3]; this is implemented in the prototype. The physical query operator *Scan-Stop* is a straightforward implementation of *Stop-After* and appears on top of the query plan given in Figure 3.

To evaluate the operators, the following performance measures are chosen:

1. The query running time to retrieve the top matching k output results.
2. The size of the buffer maintained by the operator.

3. The number of database accesses (in disk pages).

While the number of database accesses should give a good indication of the time complexity of the operator, the experiments show a significant CPU time complexity difference between the two operators that affect the total running time, especially for small numbers of inputs as shown by the following experiments. Another interesting set of experiments shows how ordering of the input streams in the pipeline affects the performance. This will have a significant impact on query optimization and the generation of query execution plans for queries involving joining multiple ranked inputs. In our experiments we study the effect of input streams ordering on both the NRA-RJ and the J^* operators.

To compare the two pipelined operators, we implement the non-pipelined version of the NRA algorithm as a multi-way rank-join operator named *MW-RJ*. Although most query optimizers are restricted to binary operators, the performance of the *MW-RJ* gives useful insight when comparing the two pipelined operators, and gives a reference line for the best possible performance to get the required results.

6.1 Real World Data Comparison

Figure 4 gives performance comparisons among the NRA-RJ (with a balancing factor $p = 2$), J^* and *MW-RJ* operators for $m = 3$, where m is the number of input sources that give a pipeline of length $m - 1$.

Figure 4(a) shows that NRA-RJ outperforms J^* in total running time, and the pipeline does not affect the speed of the NRA-RJ operator when compared with *MW-RJ*. For the maximum queue size given in Figure 4(b), the three operators have a comparable maximum queue size for small numbers of requested output objects, k . As k increases the J^* operator starts to have large queue sizes due to the fact that it has to consider all possible join combinations. Figure 4(c) shows a comparable performance of the three operators with respect to the number of database objects retrieved. Given that the number of accessed database objects of the *MW-RJ* operator is the minimum number of objects that must be visited before reporting an answer (due to the proved optimality of the NRA algorithm), Figure 4(c) shows that both the NRA-RJ and the J^* operators are close to the *MW-RJ* in the number of database accesses.

6.2 The Effect of Pipelining

In this experiment, we evaluate how scalable the two pipelined operators are with respect to the length of the query pipeline m . By fixing $k = 20$, the operators NRA-RJ and J^* are compared with respect to the three chosen performance metrics given in Section 6. Figure 5 compares the performance of the operators NRA-RJ, *MW-RJ*, and J^* as m increases from 2 to

6. Figure 5(a) shows that NRA-RJ is an order of magnitude faster than J^* with respect to the overall running time. The running time of J^* increases drastically with the increase in m making it not scalable for long query pipelines. The total running time of NRA-RJ is as good as that of MW-RJ even for long query pipelines. Figures 5(b) and (c) show that both the NRA-RJ and the J^* operators perform similarly with respect to maximum queue size and number of database accesses. The figures also show the effect of the pipelining on both operators as their performance starts to divert from that of MW-RJ as m increases.

6.3 The Effect of the Balancing Factor

The optimization proposed in Section 4 has a significant impact on the NRA-RJ performance and its scalability to long queue pipelines. In Figures 6 (a) and (b) we compare the maximum queue size and the number of accessed pages of the NRA-RJ operator for different values of p . The case in which $p = 1$ represents the unoptimized version of the NRA-RJ. The experiment shows that, for small values of k , the performance enhances as p increases. For larger values of k , increasing p results in accessing more tuples from the right child than necessary and hence small values of p gives a better performance.

We measured the effect of choosing p on the scalability of the NRA-RJ operator. Figures 5(a) and (b) give the maximum queue size and the number of accessed data pages for different values of m (the length of the pipeline). k is fixed to 20 output results. Also, we compare the performance of NRA-RJ for different values of p . When p is variable, p can have a different value in each pipeline stage. For example, $p = 1$ in the first stage and $p = 2$ in the second stage, etc. The motivation behind having different values for p in different pipeline stages is that the cost of accessing the left child increases as we go up in the query pipeline. A good heuristic is to set p to depend on the pipeline stage. The figures show that this heuristic gives the best performance for $k = 20$. Setting p to 2 enhanced the performance significantly when compared against the unoptimized version when $p = 1$.

Choosing the right p is a design decision and depends on the data and the order of the input streams, but a good choice of p boosts the performance of NRA-RJ.

6.4 The Effect of Input Ordering

In this experiment we study the effect of input stream ordering in the pipeline on the performance for both operators. Figure 8 gives the performance metrics of the NRA-RJ operator for 6 possible orderings of the input streams in a query pipeline with $m = 4$. Figure 9 gives the same metrics for the J^* operator. The results show the sensitivity of the NRA-RJ operator to the ordering of input. This sensitivity can be explained by

the excessive local ranking in the query pipeline, and hence, choosing which pairs to rank together plays a major role in getting the final results. The operator J^* is less sensitive to input orderings due to the *guided* fewer invocations of local ranking in the query pipeline. In the experiments in the previous sections, we use the ordering O_1 for both operators. Ordering O_1 shows the best performance in the case of NRA-RJ and the best execution time in the case of J^* .

7 Conclusion

In this paper, we carried out an extensive performance study to evaluate two recent algorithms for obtaining a global rank from multiple ranked inputs. The two rank-join algorithms we investigated are the J^* algorithm and an adaptation of the No-Random-Access algorithm. We focused on the use of these algorithms as binary pipelined query operators, which makes them practical for most database engines. Several experiments were conducted to illustrate the different performance issues and trade-offs. The experiments were in the context of multimedia retrieval and were performed against a continuous media retrieval prototype.

Our study shows the importance of implementing rank-join algorithms as query operators, and helps in choosing the right rank-join operator for a given problem setting. In the case where an arbitrary join condition is needed, the J^* operator is the only choice, since the NRA-RJ operator is defined only for joining multiple rankings of the same set of objects. The performance of the NRA-RJ operator is greatly enhanced through unbalancing the depth step of its inputs to reduce the overhead of local ranking in the earlier pipeline stages. As demonstrated, this optimized NRA-RJ operator is superior over the J^* operator even for large number of ranked inputs. The optimized NRA-RJ operator is an order of magnitude faster than the J^* operator, has less space requirements, and has a comparable number of disk accesses. The performance study also shows that the NRA-RJ operator is more sensitive to the ordering of the ranked input streams than the J^* operator, which shows less sensitivity. This has an impact on the optimization of rank-join queries.

Our overall conclusion is that for joining multiple rankings of the same set of objects, the NRA-RJ operator gives the best performance, while for arbitrary join conditions, the J^* operator is the best feasible choice.

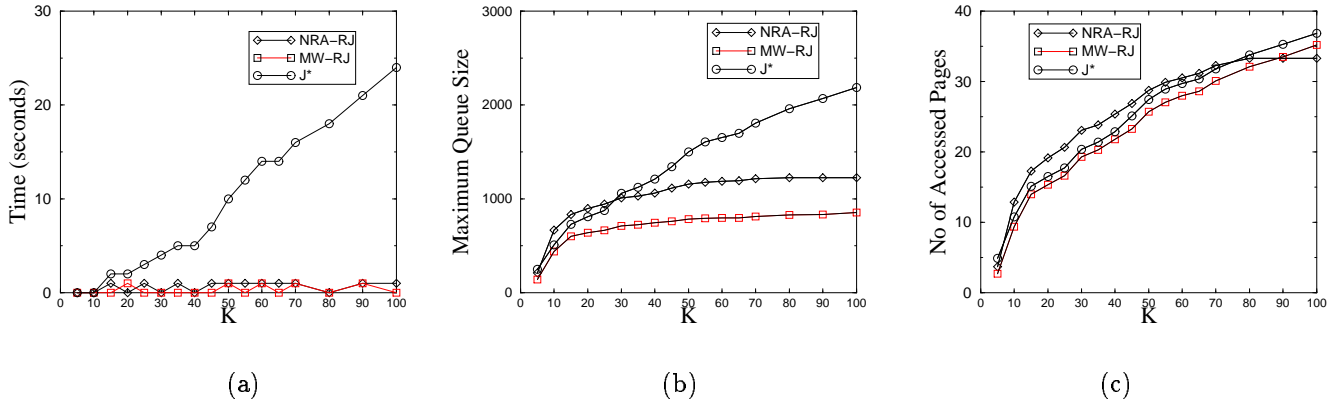


Figure 4: Comparing NRA-RJ, MW-RJ and J^* for $m = 3$.

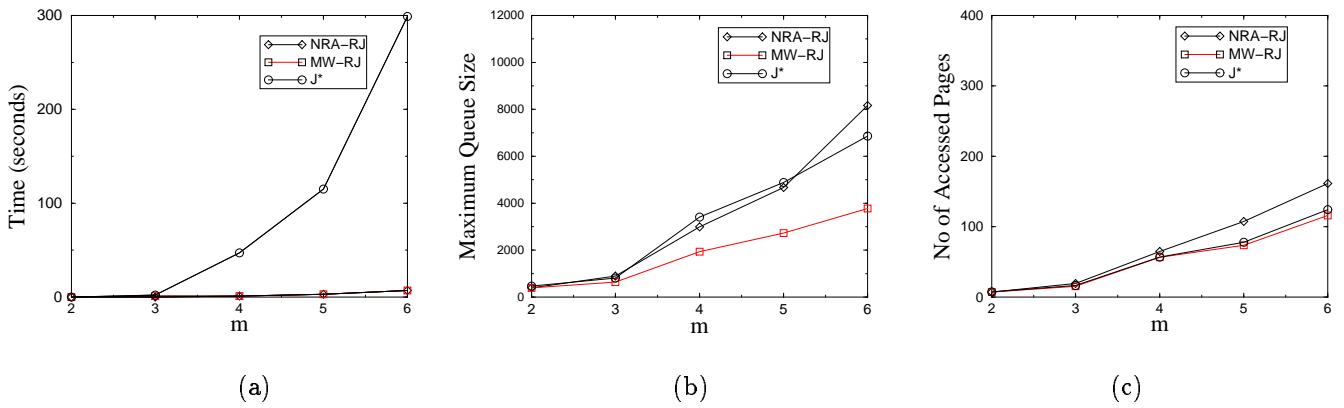


Figure 5: Scalability of the NRA-RJ operator.

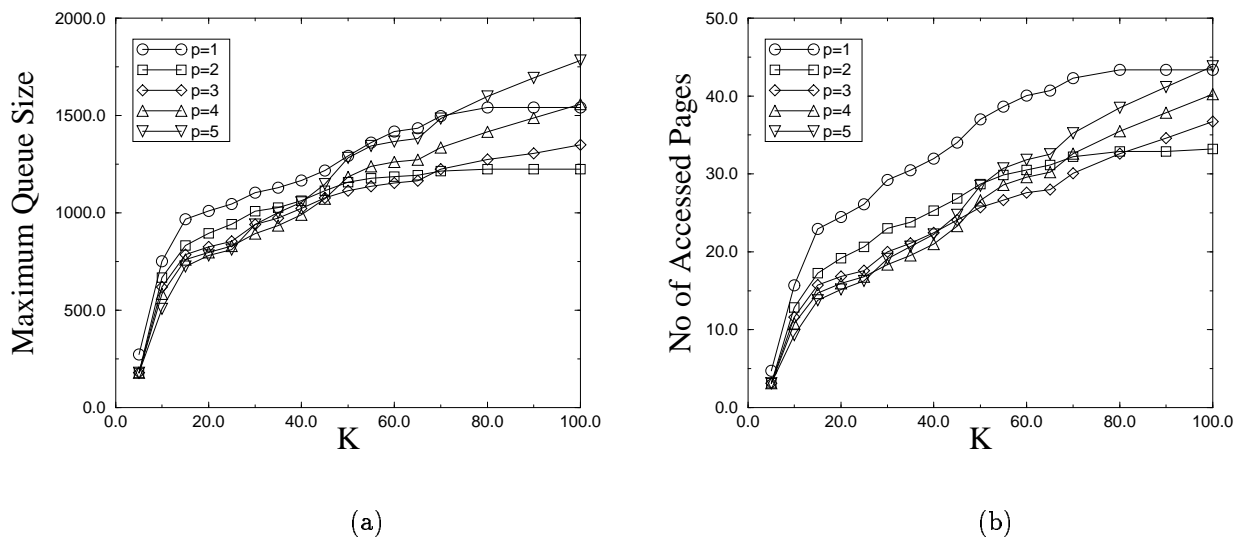
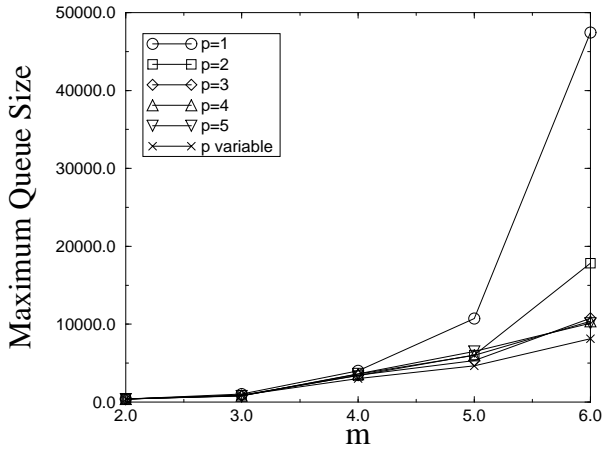
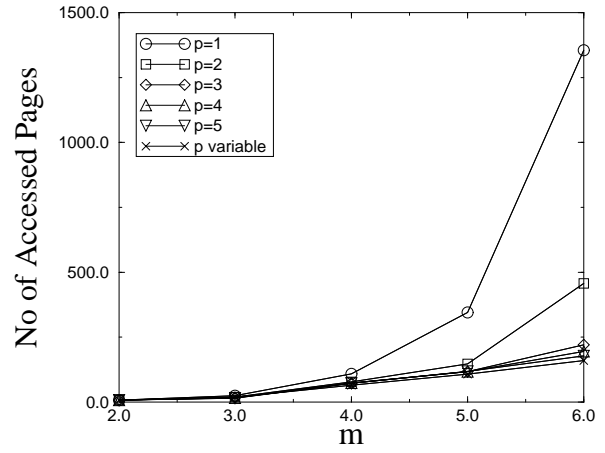


Figure 6: The effect of the balancing factor p for $m = 3$.

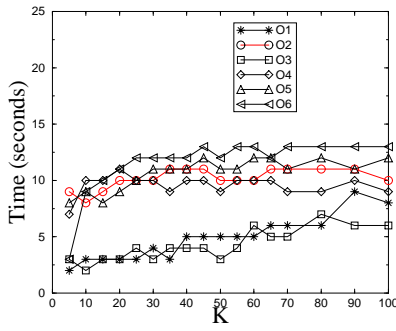


(a)

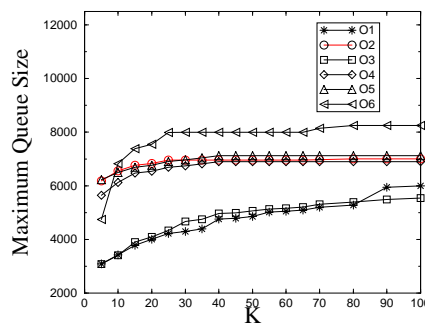


(b)

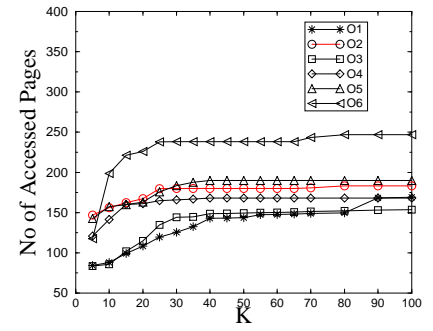
Figure 7: The effect of the balancing factor on the the scalability of NRA-RJ.



(a)

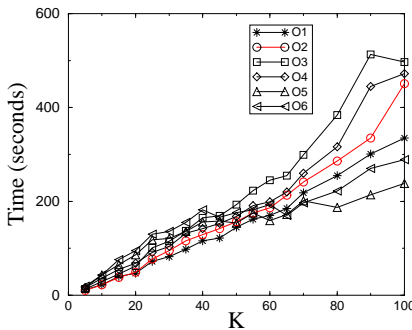


(b)

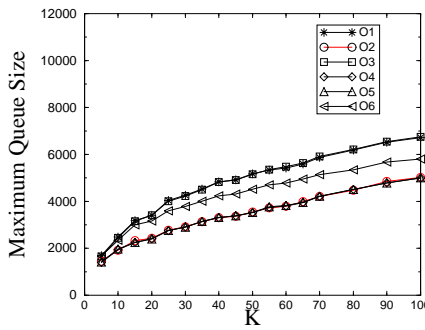


(c)

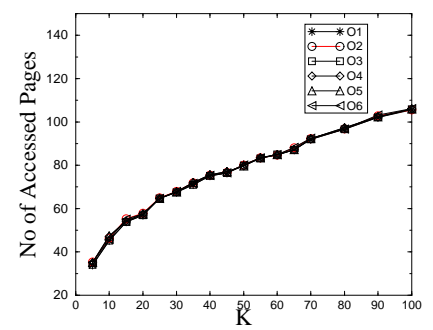
Figure 8: The effect of input stream ordering on NRA-RJ.



(a)



(b)



(c)

Figure 9: The effect of input stream ordering on J^* .

References

- [1] Storage manager architecture. *Shore documentation, Computer Sciences Department, UW-Madison*, June 1999.
- [2] Michael J. Carey and Donald Kossmann. On saying “Enough already!” in SQL. In *SIGMOD’97, Tucson, Arizona*, volume 26(2), pages 219–230, May 13–15 1997.
- [3] Michael J. Carey and Donald Kossmann. Reducing the braking distance of an SQL query engine. In *VLDB’98, New York, NY, 24–27 August, 1998*, pages 158–169, 1998.
- [4] Surajit Chaudhuri and Luis Gravano. Optimizing queries over multimedia repositories. In *SIGMOD’96, Montreal, Quebec, Canada, June 4–6, 1996*, pages 91–102, 1996.
- [5] J.Y. Chen, C. Taskiran, A. Albiol, E.J. Delp, and C.A. Bouman. Vibe: A compressed video database structured for active browsing and search. In *In Proc. SPIE: Multimedia Storage and Archiving Systems IV 3846*, pages 148–164, 1999.
- [6] Ronald Fagin. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences (JCSS)*, 58(1):83–99, Feb 1999.
- [7] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middlewar. In *PODS’2001 Santa Barbara, California, May 2001*.
- [8] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovicand D. Steele, and P. Yanker. Query by image and video content: The qbic system. In *IEEE Computer*, volume 38, pages 23–31, 1995.
- [9] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. In *VLDB 2000,, September 10–14, Cairo, Egypt*, pages 419–428, 2000.
- [10] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *In: Proceedings of the IEEE International Conference on Information Technology: Coding and Computing (ITCC 2001), Las Vegas, USA, 2001*, 2001.
- [11] A. Hamrapur, A. Gupta, B. Horowitz, C.F. Shu, C. Fuller, J. Bach, M. Gorkani, and R. Jain. Virage video engine. In *SPIE Proc. Storage and Retrieval for Image and Video Databases*, pages 188–197, 1997.
- [12] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database system. *VLDB’95*, 1995.
- [13] Norio Katayama and Shin’ichi Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2), 1997.
- [14] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. In *VLDB’01, Rome, Italy*, 2001.
- [15] Surya Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE’99, Sydney, Australia*, pages 22–29. IEEE Computer Society, 1999.
- [16] Berthold Reinwald and Hamid Pirahesh. SQL open heterogeneous data access. In *SIGMOD’98, June 2–4, 1998, Seattle, Washington, USA*, pages 506–507, 1998.
- [17] Berthold Reinwald, Hamid Pirahesh, Ganapathy Krishnamoorthy, George Lapis, Brian T. Tran, and Swati Vora. Heterogeneous query processing through sql table functions. In *ICDE’99, 23–26 March 1999, Sydney, Australia*, pages 366–373, 1999.
- [18] Mary Tork Roth, Manish Arya, Laura M. Haas, Michael J. Carey, William F. Cody, Ronald Fagin, Peter M. Schwarz, Joachim Thomas II, and Edward L. Wimmers. The garlic project. In *SIGMOD’96, Montreal, Quebec, Canada, June 4–6, 1996*, page 557, 1996.
- [19] G. Salton. *Automatic Text Processing: The Transformational, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, Reading, Massachusetts, U.S.A., 1988.
- [20] Praveen Seshadri and Mark Paskin. Predator: An or-dbms with enhanced data types. In *SIGMOD 1997, May 13–15, 1997, Tucson, Arizona, USA*, 1997.