Optimizing Queries over Multimedia Repositories

Surajit Chaudhuri*
Hewlett-Packard Laboratories
surajitc@microsoft.com

Luis Gravano
Hewlett-Packard Laboratories
Stanford University
gravano@cs.stanford.edu

Abstract

Repositories of multimedia objects having multiple types of attributes (e.g., image, text) are becoming increasingly common. A selection on these attributes will typically produce not just a set of objects, as in the traditional relational query model (*filtering*), but also a *grade of match* associated with each object, indicating how well the object matches the selection condition (*ranking*). Also, multimedia repositories may allow access to the attributes of each object only through indexes. We investigate how to optimize the processing of queries over multimedia repositories. A key issue is the choice of the indexes used to search the repository. We define an execution space that is *search-minimal*, i.e., the set of indexes searched is minimal. Although the general problem of picking an optimal plan in the search-minimal execution space is NP-hard, we solve the problem efficiently when the predicates in the query are independent. We also show that the problem of optimizing queries that ask for a few top-ranked objects can be viewed, in many cases, as that of evaluating selection conditions. Thus, both problems can be viewed together as an extended filtering problem.

1 Introduction

The problem of content management of multimedia repositories is becoming increasingly important with the development of multimedia applications. For example, digitization of photo and art collections is becoming popular, multimedia mail and groupware applications are getting widely available, and satellite images are being used for weather predictions. Attributes of the multimedia objects may include the date the multimedia object was authored, a free-text description of the object, and image features like color histograms. These attributes provide the ability to recall one or more objects from the repository. There are at least three major ways in which accesses to a multimedia repository differ from that to a structured database (e.g., a relational database). First, rarely does a user expect an *exact* match with the feature of a multimedia object (e.g., color histogram). Rather, an object does not either satisfy or fail a condition, but has instead an associated *grade* of match [1]. Thus, an atomic filter condition will not be an equality between two values (e.g., between a given color c and the color *oid.color* of an object), but instead an inequality involving the grade of match between the two values and some target grade (e.g., *Grade(color, c)(oid)* > 0.7). Next, every condition on the attributes of a multimedia object may only be evaluated through calls to an index. This is in contrast to a traditional database where, after accessing a tuple, all selection predicates can be evaluated on the tuple. Finally, the process of querying and browsing over a multimedia repository is likely to be interactive, and users will tend to ask for only a few best matches according to a ranking criterion.

The above observations lead us to investigate a query model with *filter conditions* as well as *ranking expressions*, and to study the cost-based optimization of such queries. In general, a query will specify both a filter condition F and a ranking expression R. The query answer is a rank of the objects that satisfy F, based on their grade of match for the ranking expression R.

Optimizing a filter condition in this querying model presents new challenges. An atomic condition can be processed in two ways: by a *search*, where we retrieve all the objects that match the given condition (access by value), and by a *probe*, where instead of using the condition as an access method, we only test it for each (given) object id (access by object id).

The costs of these two kinds of accesses, search and probe, in multimedia repositories can vary for a single data and attribute type as well as across types. How to order a sequence of probes without considering the search costs, as well as how to determine a set of search conditions when the probing cost is zero (or a

*Currently with Microsoft Research.

constant) has been studied before. However, to the best of our knowledge, no work has studied the optimization problem when both searches and probes have non-zero costs and the filter condition is an arbitrary boolean expression. When the filter condition is a conjunction of atomic conditions, the problem becomes closely related to that of ordering joins.

To optimize the processing of a filter condition, we define a space of *search-minimal executions*, and show how to pick the optimal execution in that space when the conditions in the filter condition are *independent*. Our experiments show that considering both the search and probe costs during query optimization impacts the choice of an execution plan significantly. We also prove that if the conditions in the filter condition are not independent, the problem of determining an optimal search-minimal execution is NP-hard. Although the search-minimal execution space is a restricted space, our experiments indicate that a simple post-optimization step leads to plans that are nearly always as good as the plans obtained when plans are not restricted to be search minimal [2].

Our paper also contributes to the problem of optimizing the evaluation of a ranking expression. Previous significant work in this area is due to Fagin [1], who shows his algorithm to be asymptotically optimal under broad assumptions. A key contribution of our paper is to show that ranking expressions can be processed "almost" like filter conditions. We prove that our technique is expected not to retrieve more objects than the strategy in [1]. Our experimental results indicate that the performance gain from processing ranking expressions as filter conditions can be substantial. This result allows us to process queries with both a filter condition and a ranking expression in a unifying framework.

The rest of the paper is organized as follows. Section 2 describes the query model that we use. Sections 3 and 4 present the results on evaluating filter conditions and ranking expressions, respectively. Section 5 discusses our experimental results. Finally, Section 6 is devoted to related work.

2 Query Model

In this section we introduce a query model to *select* multimedia objects from a repository. (See [3] for a similar model.) Such a query model needs to satisfy the following requirements:

1. Consider that a match between the value of an attribute of a multimedia object and a given constant is *not* exact, i.e., must account for the grade of match.
2. Allow users to specify thresholds on the grade of match of the acceptable objects.
3. Enable users to ask for only a few top-matching objects.

Given an object o, an attribute *attr*, and a constant *value*, the notion of a *grade of match Grade(attr, value)(o)* between o and the given *value* for attribute *attr* addresses the first requirement. Such a grade is a real number in the $[0, 1]$ range and designates the degree of equality (match) between *o.attr* and *value*.

We address the second requirement by introducing the notion of a *filter condition*. The *atomic* filter conditions are of the form $Grade(attr,\ value)(o) \geq grade$. An object o satisfies this condition if the grade of match between its value *o.attr* for attribute *attr* and constant *value* is at least *grade*. Additional filter conditions are generated from the atomic conditions by using the $\wedge$ ("and") and $\vee$ ("or") boolean connectives. Filter conditions evaluate to either true or false.

Following [1], we address the third requirement for the query model through the notion of a *ranking expression*. The ranking expression computes a *composite grade* for an object from individual grades of match and the composition functions *Min* and *Max*. Every object has a grade between 0 and 1 for a given ranking expression. Users can use a ranking expression in their queries, and ask for k objects with the top grades for the given ranking expression.

We use the following SQL-like syntax to describe the queries in our model:

```
SELECT oid
FROM Repository
WHERE Filter_condition
ORDER [k] by Ranking_expression
```

The above query asks for k objects in the object repository with the highest grade for the ranking expression, among those objects that satisfy the filter condition. The filter condition eliminates unacceptable matches, while the ranking expression orders the acceptable objects.

Example 2.1: *Consider a multimedia repository of information on criminals. A record on every person on file consists of a textual description* `p` *(for profile), a scanned fingerprint* `f`*, and a recording of a voice sample* `v`*. Given a target fingerprint* `F` *and voice sample* `V`*, the following example asks for records (1) whose fingerprint matches* `F` *with grade 0.9 or higher, or (2) whose profile matches the string* `'on parole'` *with grade 0.9 or higher and whose voice sample matches* `V` *with grade 0.5 or higher. The ranking expression ranks the acceptable records by the maximum of their grade of match for the voice sample* `V` *and for the fingerprint* `F`*. The answer contains the top 10 such acceptable records. (For simplicity, we omitted the parameter* `oid` *in the atomic conditions below.)*

```
SELECT oid
FROM   Repository
WHERE  (Grade(v, V) >= .5 AND
        Grade(p, 'on parole') >= .9)
       OR Grade(f, F) >= .9
ORDER [10] BY Max(Grade(f, F), Grade(v, V))
```

2.1 Expressivity of the Query Model

The filter condition F in a query Q selects the set of objects in the repository that satisfy the condition, whereas the ranking expression R computes a grade for each qualifying object. We use these grades for ordering the objects that satisfy the filter condition. An interesting expressivity question is whether we actually need both F and R. In other words, we would like to know whether we can "embed" the filter condition F in a new ranking expression R_F such that the top objects according to R_F are the top objects for R that satisfy F. (Note that a filter condition does not impose an order on the objects, therefore we cannot express R and F using a single filter condition F_R. However, see Section 4.)

More formally, given F and R, a ranking expression R_F that replaces F and R should verify the following two conditions for any database db and for any given k, assuming that at least k objects satisfy F in database db:

1. An object $o \in db$ is among the top k objects according to R_F only if o satisfies F.
2. If objects o, $o' \in db$ satisfy F, and $R(o) < R(o')$, then $R_F(o) < R_F(o')$.

The following example establishes the need for both a filter condition and a ranking expression in our model. It shows that it is not possible to find such a ranking expression R_F for an arbitrary filter condition F and an arbitrary ranking expression R.

Example 2.2 : *Let $e_1 = Grade(a_1, v_1)$ and $e_2 = Grade(a_2, v_2)$, where a_1 and a_2 are different attributes, and v_1 and v_2 are constants. Consider the filter condition $F = e_1 \geq 0.2$, and the ranking expression $R = e_2$. The query associated with F and R ranks the objects that have grade 0.2 or higher for e_1 according to their grade for e_2.*

Suppose that there is a ranking expression R_F that satisfies the two conditions above. Then, R_F is equivalent to (i.e., always produces the same grades as) one of the following expressions: e_1, e_2, $Min(e_1, e_2)$, or $Max(e_1, e_2)$. Consider the database of three objects described in Table 1, and that we are interested in the top object ($k = 1$) for R that satisfies F. The actual answer to the query should be object o_2, which has the highest grade for R (0.4) among the two objects (o_2 and o_3) that pass the filter condition F. However, each of the four possibilities for R_F produces a wrong answer for the query (either o_1 or o_3).

Object	e_1	e_2	$Min(e_1, e_2)$	$Max(e_1, e_2)$
o_1	0.1	0.6	0.1	0.6
o_2	0.2	0.4	0.2	0.4
o_3	0.5	0.3	0.3	0.5

Table 1: The three objects in the database, and their grades for each of the four possible definitions of R_F.

2.2 Storage Level Interfaces

A repository has a set of multimedia objects. We assume that each object has an id and a set of attribute values, which we can only access through indexes. Given a value for an attribute, an index supports access to the ids of the objects that match that value with a certain grade, as we discuss below. Indexes also support access to the attribute values of an object given its oid.

The following are several storage-level access interfaces that multimedia repositories may support [4]. Key to these interfaces is that the objects match attribute values with a grade of match:

- *GradeSearch(attribute, value, min_grade)*: Given a value for an attribute, and a minimum grade requirement, returns the set of objects that match the attribute value with at least the specified grade, together with the grades for the objects.
- *TopSearch(attribute, value, count)*: Given a value for an attribute, and the count of the number of objects desired, returns a list of *count* objects that match the attribute value with the highest grades in the repository, together with the grades for the objects.
- *Probe(attribute, value, {oid})*: Given a set of object ids and a value for an attribute, returns the grade of each of the specified objects for the attribute value.

Not all repositories have to support all of these interfaces at the physical level. For example, a repository may implement *Probe* atop *GradeSearch*. Next, we briefly describe how text and image attributes may support the above interfaces.

Text Attributes:

Consider a repository of objects with a textual attribute T. For this attribute, the repository might have an index that handles queries using the *vector-space* model of document retrieval [5]. Given a value for T (i.e., a sequence of words), this index assigns a grade to every object in the repository, according to how *similar* its value for T and the query value are. Vector-space retrieval systems usually provide the *GradeSearch* interface, the *TopSearch* interface, or both. Some text-retrieval systems allow access to the document weight vectors by document id. If this is the case, the *Probe* interface is readily provided by accessing the weight vectors of the objects requested, and computing the similarity of these vectors and the query vector. If this direct access is not provided, *Probe* can be simulated by *GradeSearch* or *TopSearch* by requesting all objects with non-zero similarity, for example.

Image Attributes:

If the objects of a repository contain an image, an attribute could be the color histogram of this image, for example. Then, a filter condition on such an attribute may ask for objects whose color histogram matches a given color histogram closely. The QBIC system supports this type of queries [4]. A popular data structure to support such queries is the R tree [6] and its variants [7, 8], which may be used to index the feature vectors associated with the attributes. The grade of match between two feature vectors is computed based on the semantics of the attributes.

Given one feature-vector attribute, a value v for the attribute, and a grade, *GradeSearch* can be implemented over an R tree by determining a box around the given value v that contains all vectors that match v with the given grade or higher, for a given grade-computation algorithm. We then process the corresponding range search. Roussopoulos and others [9] have recently presented an algorithm to find nearest neighbors on R trees. This algorithm can be used for implementing *TopSearch*.

3 Filter Conditions

In this section we will consider the processing and cost-based optimization of queries that have only a filter condition, i.e., they are of the form:

```
SELECT oid
FROM Repository
WHERE Filter_condition
```

We will assume that the filter conditions are *independent*. Similar restrictions have been traditionally adopted [10].

Definition 3.1: *We say that a filter condition f is* independent *if:*

- *Every atomic filter condition occurs at most once in f.*
- *Every n atomic filter conditions $e_1, \ldots, e_n$ satisfy $p(e_1 \wedge \ldots \wedge e_n) = \Pi_{i=1}^{n} p(e_i)$, where $p(e)$ is the probability that the filter condition e is true.*

We assume that our repository requires that we use an index to evaluate every atomic filter condition. One way to process such queries is to retrieve object ids using one *GradeSearch* for *each* atomic condition in the filter condition, and then merge these sets of object ids through a sequence of unions and intersections.

Alternatively, we can retrieve a set of object ids using *GradeSearch* for *some* conditions. The key optimization problem is to determine the set of filter conditions that are to be evaluated using *GradeSearch*. The rest of the conditions will be evaluated by using *Probe*. In order to efficiently execute the latter step, we will exploit the known techniques in optimizing the probing of expensive filter conditions [11, 12, 13, 14].

In this section, we first define a space of *search-minimal* executions, and sketch the cost model and the optimization criteria. Next, we describe an optimization algorithm and explain the conditions under which it is optimal. We conclude with a result that indicates that the general problem of determining an optimal search-minimal execution is NP-hard. The results in this section are complemented by the experiments in Section 5.1, which show that considering both the search and probe costs leads to significantly better execution strategies.

3.1 Execution Space

We begin by discussing the possible space of execution for simple filter conditions, i.e., conditions that consist of a disjunction (or a conjunction) of atomic conditions. We will then generalize our description for arbitrary filter conditions with disjunctions and conjunctions.

To process an atomic condition *Grade(attr, value)(o)* $\geq$ *grade*, we use the *GradeSearch(attr, value, grade)* access method described in the previous section.

Consider now the case where the filter condition is a disjunction of atomic filter conditions $a_1 \vee \ldots \vee a_n$[1]. All objects that satisfy at least one of the a_i satisfy the entire filter condition. Evaluation of an atomic condition a_i requires the use of the *GradeSearch* access method associated with a_i. Since we assume that the atomic conditions are independent, use of a *GradeSearch* is needed for each atomic condition not to miss any object that satisfies the entire condition.

Consider now the case where the filter condition is a conjunction of atomic filter conditions $a_1 \wedge \ldots \wedge a_n$. There are several execution alternatives. In particular, we can retrieve all the objects that may satisfy the filter condition by using *GradeSearch* on any of the atomic conditions $a_1, \ldots, a_n$. Subsequently, we can test each retrieved object to verify that it satisfies all of the remaining conditions. The cost of using one atomic condition for *GradeSearch* instead of another may lead to significant differences in the cost. Thus, we can process a conjunction of atomic filter conditions by executing the following steps:

[1] We use a_j as a shorthand for an atomic condition specifying an attribute, value, and grade, e.g., *Grade(attr,val)(o)* $\geq$ *grade*.

1. *Search:* Retrieve objects based on one atomic condition (using *GradeSearch*).
2. *Probe:* Test that the retrieved objects satisfy the other conditions (using *Probe*).

An important optimization step is to carry out Step (2) efficiently by ordering the atomic-condition probes (Section 3.3).

We call the above class of execution alternatives for a conjunctive query *search-minimal* since only a minimal set of conditions (in this case, only one condition) is used for *GradeSearch*; the other conditions are evaluated using *Probe*. The search-minimal strategies represent a subset of the possible executions. In particular for a conjunctive filter condition, instead of searching on a single atomic condition and probing on the others, it is possible to search on any subset of the atomic conditions and to take the intersection of the sets of object-ids retrieved. However, the space of all such executions is significantly larger. In [2] we consider applying a post-optimization step to the best search-minimal strategy and compare the new strategy against the overall optimal execution. (Neither the post-optimized strategy nor the overall optimal one are necessarily search minimal.) The post-optimization step produces almost optimal strategies most of the time [2].

By searching on a condition using *GradeSearch*, we obtain a set of objects. However, we may need to do additional probes to determine the subset of objects that satisfy the rest of the filter condition as well. Thus, given an atomic condition a_i and a filter condition f, the *residue* of f for a_i, $R(a_i, f)$, is a boolean condition that the objects retrieved using a_i should satisfy to satisfy the entire condition f. The following definition captures how we construct residues for independent filter conditions.

Definition 3.2: *Let f be an independent filter condition, represented as a tree, and a an atomic condition of f. Consider the path from the leaf node for (the only occurrence of) a to the root of the tree for f. For every $\wedge$ node i in this path, let α_i be the condition consisting of the conjunction of all the subtrees that are children of the node i and that do not contain a. Then the* residue *of f for a, $R(a, f)$, is $\bigwedge_i \alpha_i$. If there are no such nodes, then $R(a, f) = true$.*

Example 3.3: *Consider the filter condition $f = a_4 \wedge ((a_1 \wedge a_2) \vee a_3)$. Let us consider the residue of the atomic condition a_2 using the definition above. Thus, $\alpha_1 = a_1$ and $\alpha_2 = a_4$. Hence, $R(a_2, f) = a_1 \wedge a_4$. As another example, $R(a_4, f) = (a_1 \wedge a_2) \vee a_3$. Then, any object that satisfies a_4 and also satisfies $R(a_4, f)$ satisfies the entire condition f.*

Given a filter condition f, we would like to characterize the smallest sets of atomic conditions such that by searching the conditions in any of these sets we retrieve all of the objects that satisfy f (plus some extra ones that are pruned out by probing).

Definition 3.4: *A* complete *set of atomic conditions m for a filter condition f is a set of atomic conditions in f such that any object that satisfies f also satisfies at least one of the atomic conditions in m. A complete set m for f is a* search-minimal condition set *for f if there is no proper subset of m that is also complete for f.*

Example 3.5: *Consider Example 3.3. Each of $\{a_4\}$, $\{a_2, a_3\}$, and $\{a_1, a_3\}$ is a search-minimal condition set. If we decide to search on $\{a_2, a_3\}$, the following three steps yield exactly all of the objects that satisfy f:*

1. *Search on a_2 and probe the retrieved objects with residue $R(a_2, f) = a_1 \wedge a_4$. Keep the objects that satisfy $R(a_2, F)$.*
2. *Search on a_3 and probe the retrieved objects with residue $R(a_3, f) = a_4$. Keep the objects that satisfy $R(a_3, F)$.*
3. *Return the objects kept.*

Proposition 3.6: *Let m be a complete set of atomic conditions for an independent filter condition f. Then, $f \equiv \bigvee_{a \in m}(a \wedge R(a, f))$. In particular, the above holds if m is a search-minimal condition set for f.*

Now we are ready to define the space of search-minimal executions.

Definition 3.7 : *A* search-minimal execution *of an independent filter condition f searches the repository using a search-minimal condition set m for f, and executes the following steps:*

- *For each condition $a \in m$:*
  - *Search on a to obtain a set of objects S_a.*
  - *Probe every object in S_a with the residual condition $R(a, f)$ to obtain a filtered set S'_a of objects that satisfy f.*
- *Return the union $\bigcup_{a \in m} S'_a$.*

We now present algorithms to pick a plan from the space of search-minimal executions.

3.2 Assumptions and the Cost Model

Our optimization algorithm is cost-based and makes statistical assumptions about the query conditions as well as about the availability of certain statistical estimates. We describe these assumptions in this section.

We associate the following statistics with each atomic condition a. We assume that we can extract these statistics from the underlying object repository and its indexes.

- *Selectivity Factor $Sel(a)$:* Fraction of the objects in the repository that satisfy the condition a.

- *Search Cost $SC(a)$:* Cost of retrieving the ids of the objects that satisfy the condition a using *GradeSearch*.

- *Probe Cost $PC(a,p)$:* Cost of checking the condition a for p objects, using the *Probe* access method.

The probe cost $PC(a,p)$ depends on p, the number of probes that need to be performed. If p is large enough, it might be cheaper to implement the p probes by doing a single search on a, at cost $SC(a)$. This observation is the key of the post-optimization step in [2].

We now sketch how to estimate the cost parameters over multimedia repositories for text and image attributes. Consider first a text attribute that is handled by a vector-space retrieval system. Typically, such a system has inverted lists associated with each term in the vocabulary [5]. For each term we can extract the number of documents d that contain the term, and the added weight w of the term in the documents that contain it. Thus, we can use the methodology in [15] to estimate the selectivity of an atomic filter condition, as well as the cost of processing the inverted lists that the condition requires.

Consider now an attribute over an image that is handled with an R tree. We can then use the methodology in [16], which uses the concept of the fractal dimension of a data set to estimate the selectivity of atomic conditions, and the expected cost of processing such conditions using the R tree. We use this estimation technique for our experiments.

We will restrict our discussion to optimizing independent filter conditions containing disjunctions and conjunctions. We can compute the selectivities of complex independent filter conditions using the following two rules as in traditional optimization [10]:

- $Sel(e_1 \wedge \ldots \wedge e_n) = \Pi_{i=1}^{n} Sel(e_i)$

- $Sel(e_1 \vee \ldots \vee e_n) = 1 - \Pi_{i=1}^{n}(1 - Sel(e_i))$

3.3 Optimization Algorithm

In this section, we present the results on optimization of filter conditions. First, we define our optimization metric over the search-minimal execution space. Next, we sketch how we can use the past work in optimizing boolean expressions for the task of determining a strategy for probing. Then, we present our algorithm, which is optimal for independent filter conditions, and discuss how we can adapt it for non-independent filter conditions. We conclude with an NP-hardness result that shows that if the filter condition is not independent, then the complexity of determining an optimal execution is NP-hard.

Cost of the Search Minimal Executions:

Given a search-minimal condition set m for a filter condition f and an algorithm w for probing conditions, we define $C_w(f,m)$, the cost of searching the conditions in m plus the cost of probing the other conditions using algorithm w, as follows:

$$C_w(f,m) = \sum_{a \in m} (SC(a) + PC_w(R(a,f), |O_a|))$$

where $|O_a|$ is the number of objects that satisfy condition a and $PC_w(R(a,f),|O_a|)$ is the cost of probing condition $R(a,f)$ for $|O_a|$ objects using algorithm w. This cost depends on the probing algorithm w, as we discuss next. Note that if there are O objects in the repository, $|O_a| = Sel(a) * O$.

Optimizing Evaluation of Residues:

Given a residue $R(a,f)$, the task of determining an optimal evaluation for $R(a,f)$ maps to the well studied problem of optimizing the execution of selection conditions containing expensive predicates [11]. (See also [12, 13, 14].)

If $R(a,f)$ is a conjunction of atomic conditions $a_1 \wedge \ldots \wedge a_n$ with $n > 1$, there is an efficient algorithm w that finds the optimum probing strategy. We first order the atomic conditions in increasing *rank* order, where the rank of the condition a_i is $\frac{Sel(a_i)-1}{c_i}$, assuming that $PC(a_i,p) = c_i * p$ for some constant c_i. Then, given p, we can calculate the cost $PC(R(a,f),p)$ as follows. For simplicity, we assume that $a_1 \ldots a_n$ represents the increasing rank ordering of the conjuncts: $PC(R(a,f),p) = \sum_{i=1}^{n} S_i$, where $S_i = Sel(a_1) * \ldots * Sel(a_{i-1}) * p * c_i$. This result is well known and was observed in the database context by [13, 14]. We can take a similar approach to order the evaluation of a disjunction of atomic conditions.

In case $R(a,f)$ is an arbitrary boolean condition, the problem of evaluating it optimally is known to be intractable. However, several good heuristics are available [11]. Therefore, we assume that we exploit one of these available algorithms to optimize the evaluation of residues. As we mentioned above, depending on the strategy w used to evaluate $R(a,f)$, we can parameterize our cost function. Thus, we denote the cost corresponding to evaluation strategy w by C_w. However, for the rest of the discussion, we assume that such a choice of w is implicit and therefore omit references to w.

Optimality:

We now study how to find an optimal search-minimal condition set for a filter condition.

Definition 3.8: *Let f be an independent filter condition and let $M(f)$ be the set of all search-minimal con-*

dition sets for f. A search-minimal condition set m for f is optimal *if* $C(f, m) = \min_{m' \in M(f)} C(f, m')$

The algorithm to determine an optimal search-minimal condition set for an independent filter condition is implicit in the following inductive definition. Intuitively, the algorithm traverses the condition tree in a bottom-up fashion to create an optimal condition set.

Definition 3.9: *Let f be a filter condition and f' be a subexpression of f. The* inductive *search-minimal condition set for f' with respect to f, $SM_f(f')$, is defined inductively as follows:*

1. *Case $f' = a$: $SM_f(f') = \{a\}$, where a is an atomic condition*
2. *Case $f' = f_1 \wedge \ldots \wedge f_n$: $SM_f(f') = SM_f(f_i)$, where $C(f, SM_f(f_i)) = \min\{C(f,\ SM_f(f_1)),\ \ldots,\ C(f,\ SM_f(f_n))\}$ (Break ties arbitrarily.)*
3. *Case $f' = f_1 \vee \ldots \vee f_n$: $SM_f(f') = SM_f(f_1) \cup \ldots \cup SM_f(f_n)$*

Theorem 3.10: *Let f be an independent filter condition. Then $SM_f(f)$ is an optimal search-minimal condition set for f.*

The proof of optimality of $SM_f(f)$ (see [2]) depends on the fact that the given filter condition f is independent. Nonetheless, we can easily modify the above algorithm to provide a search-minimal condition set when the given filter condition is not independent. However, this set is no longer guaranteed to be optimal [2]. This is not surprising given that the general optimality problem is intractable, as the following theorem shows.

Theorem 3.11: *The problem of determining an optimal search-minimal condition set for an arbitrary filter condition is NP-hard.*

4 Filter Conditions and Ranking Expressions

In this section, we consider queries consisting not only of a filter condition, but also of a ranking expression. The answer to such queries is the top objects ordered by the ranking expression that also satisfy the filter condition. We first look at queries consisting only of ranking expressions. Section 4.1 describes an algorithm for processing this type of queries that has been recently presented [1]. Section 4.2 presents our main result regarding this class of queries. We show that we can map a given ranking expression into a filter condition, and process the ranking expression "almost" as if it were a filter condition. This result is central to processing queries with ranking expressions using the techniques of Section 3 for filter conditions. The experimental results of Section 5.2 show that the number of objects retrieved when processing a ranking expression like a filter condition can be considerably smaller than when processing the ranking expression using the algorithm in [1].

A query consisting of only a ranking expression has the form:

```
SELECT oid
FROM Repository
ORDER [k] by Ranking_expression
```

The result of this query is a list of k objects in the repository with the highest grade for the given ranking expression. The ranking expressions are built from atomic expressions that are combined using the *Min* and *Max* operators that we defined in Section 2.

4.1 Fagin's Strategy

Recently, Fagin presented a novel approach to processing a query consisting of a ranking expression [1]. In this section we briefly describe his approach.

Consider a ranking expression $R = Min(a_1, \ldots, a_n)$, where the a_i's are independent atomic expressions. Suppose that we are interested in k objects with the highest grades for R. Fagin's algorithm uses the *TopSearch* access method to retrieve these objects from the repository. He does so by retrieving the top objects from each of the atomic expressions a_i, $i = 1, \ldots, n$, until there are at least k objects in the intersection of the n streams of objects that he retrieves. (There is one stream for each a_i.) He proved that the set of objects that he retrieved contains the necessary k top objects. Therefore, he can compute the final grade for R of each of the objects retrieved, doing the necessary probes, and output the k objects with the highest grades.

Fagin has proved the important result that his algorithm to retrieve k top objects for an expression R that is a *Min* of independent atomic expressions is asymptotically optimal with arbitrarily high probability in terms of the number of objects retrieved.

Assuming independence of the subexpressions, the expected number of objects that are retrieved from each of the expressions $a_1, \ldots, a_n$ is $L = k^{\frac{1}{n}} \cdot O^{1-\frac{1}{n}}$, where O is the number of objects in the repository. In other words, a request for top k objects for R results in an expected top L object retrievals over each atomic expression $a_1, \ldots, a_n$. When the ranking expression is $R = Max(a_1,\ \ldots,\ a_n)$, Fagin's algorithm requests exactly k objects from each atomic expression a_i. It follows that there are k top objects for R among these $k \cdot n$ objects.

4.2 Processing Ranking Expressions as Filter Conditions

In this section we show that we can process ranking expressions like a modified filter condition, providing for a uniform treatment of ranking expressions and filter conditions.

Given a ranking expression R and the number k of objects desired, we show that:

1. There is an algorithm to assign a grade to each atomic expression in R, and a filter condition F with the same "structure" as R, such that F is expected to retrieve at least the top k objects according to R.

2. There is a search-minimal execution for F that retrieves an expected number of objects that is no larger than the expected number of objects that Fagin's algorithm would retrieve for R and k.

Example 4.1 : *Consider a ranking expression* $e = Min(Grade(a_1, v_1), Grade(a_2, v_2))$*, where* a_i *is an attribute, and* v_i *a constant value. We want two objects with the top grades for* e*. Now, suppose that we can somehow find a grade* G *(the higher the better) such that there are at least two objects with grade* G *or higher for expression* e*. Therefore, if we retrieve all of the objects with grade* G *or higher for* e*, we can simply order them according to their grades, and return the top two as the result to the query.*

In other words, we can process e *by processing the following associated* filter condition f*, followed by a sorting step of the answer set for* f*:*

$$f = (Grade(a_1, v_1)(o) \geq G) \wedge (Grade(a_2, v_2)(o) \geq G)$$

By processing f *using the strategies in Section 3, we obtain all of the objects with grade* G *or higher for* a_1 *and* v_1*, and for* a_2 *and* v_2*. Therefore, we obtain all of the objects with grade* G *or higher for the ranking expression* e*. If there are enough objects in this set (i.e., if there are at least two objects), then we know we have retrieved the top objects that we need to answer the query with ranking expression* e*. Similarly, we can process a ranking expression* $e' = Max(Grade(a_1, v_1), Grade(a_2, v_2))$ *as filter condition* $f' = (Grade(a_1, v_1)(o) \geq G') \vee (Grade(a_2, v_2)(o) \geq G')$*, for some grade* G'*.*

The example above shows how we can process a ranking expression e as a filter condition f followed by a sorting step. But the key point in mapping the ranking problem to a (modified) filtering problem is finding the grade G to use in f.

We now present the algorithm `Grade_Rank`, which given the number of objects desired k, a ranking expression e, and selectivity statistics, produces the grade G for the filter condition f.

1. Propagate the number of objects requested k down to each atomic expression. Assign to each atomic expression e_i the number of objects L_i that Fagin's algorithm is expected to retrieve using e_i.

2. Replace each L_i by a grade g_i using selectivity statistics: the expected number of objects having grade at least g_i for atomic expression e_i is at least L_i.

3. Propagate the g_i grades from the atomic expressions up to the entire ranking expression. A subexpression $Min(t_1, \ldots, t_n)$ gets the minimum of the grades assigned to its subexpressions t_i. A subexpression $Max(t_1, \ldots, t_n)$ gets the maximum of the grades assigned to its subexpressions t_i.

4. Return the grade G assigned to the entire ranking expression.

Example 4.2: *Consider the expression* $e = Min(e_1, e_2)$ *of Example 4.1, where* $e_i = Grade(a_i, v_i)$*. Suppose that the number of objects in the repository is* $O = 100$*. We want a top object for* e*, i.e.,* $k = 1$*. The expected number of objects that Fagin's algorithm will retrieve from both* e_1 *and* e_2 *is* $L_1 = L_2 = (k * O)^{\frac{1}{2}} = 10$*. Suppose that the largest grade* g_1 *such that* $Sel(e_1, g_1) \geq \frac{L_1}{O}$ *is* $g_1 = 0.2$*. Similarly,* $g_2 = 0.3$*. Then, the algorithm above propagates these two grades to the whole expression* e*, yielding* $G = \min\{0.2, 0.3\} = 0.2$*. We use this value of* G *in the filter condition* f *with which we will process* e*:*

$$f = (Grade(a_1, v_1)(o) \geq 0.2) \wedge (Grade(a_2, v_2)(o) \geq 0.2)$$

Once we have determined the grades g_i for the atomic expressions, we obtain a single grade G that we use in the filter condition for processing the ranking expression. The following example illustrates why we need to find G, instead of just using the g_i's.

Example 4.2: (cont.) *Suppose that instead of* f *we used the following filter condition* f_w *(for "wrong") for processing* e*:*

$$f_w = (Grade(a_1, v_1)(o) \geq 0.2) \wedge (Grade(a_2, v_2)(o) \geq 0.3)$$

The only difference between f *and* f_w *is that in* f_w *we kept the original* g_i *grades instead of using the "global" grade 0.2. Consider a database with two objects* o_1 *and* o_2*, with* $Grade(a_1, v_1)(o_1) = 0.2$ *and* $Grade(a_2, v_2)(o_1) = 0.3$*, and* $Grade(a_1, v_1)(o_2) = 0.25$ *and* $Grade(a_2, v_2)(o_2) = 0.25$*. Object* o_1 *satisfies condition* f_w*, and will thus be retrieved by* f_w*. Also,* $e(o_1) = \min\{0.2, 0.3\} = 0.2$*. On the other hand, object* o_2 *will not be retrieved by* f_w*, since it fails to satisfy the second conjunct of* f_w*. However,* $e(o_2) = \min\{0.25, 0.25\} = 0.25$*. Therefore,* o_2 *is better than* o_1 *for* e*, and yet it was not retrieved by the filter condition* f_w*.*

If we use f *instead of* f_w*, both* o_1 *and* o_2 *are retrieved. After the objects are retrieved, they are sorted according to their grade for* e*, and the best object,* o_2*, is returned which is a correct answer.*

Now we show that if we process a ranking expression (and its associated number of objects requested k) by using a filter condition F with grade G as determined by algorithm `Grade_Rank`, we can expect to retrieve no more objects than Fagin's algorithm, under some assumptions on the repositories. This result allows us to translate the ranking expressions into filter conditions, and to use the processing strategy of Section 3. At least k objects are expected to satisfy F. However, if at run time we find that fewer than k objects satisfy F, we should lower the grade G used in F. We will investigate strategies to lower G as part of our future work.

Definition 4.3: *A repository is* uniquely graded *if for every atomic expression e and integer L there exists a grade g such that $Sel(e, g) \times O = L$, where O is the number of objects in the repository.*

Intuitively, a uniquely graded repository does not have "ties": given an atomic expression e and a number L, there is a grade cut-off for the expression such that there are exactly L objects with such a grade or higher for e.

The following theorem follows for any uniquely graded repository:

Theorem 4.4 : *Let R be an independent ranking expression over a uniquely graded repository with O objects, and k be the number of objects desired. Let F be the filter condition associated with R that uses the grade G computed by algorithm* `Grade_Rank` *for R and k. Then:*

- *$Sel(F) \times O \geq k$ (i.e., at least k objects are expected to satisfy F), and*
- *There is a search-minimal execution for F such that it is expected to retrieve no more objects than Fagin's strategy is.*

Example 4.2: (cont.) *Suppose that $Sel(e_1, 0.2) = Sel(e_2, 0.3) = \frac{L}{O} = 0.1$. Then, $Sel(f) \geq 0.1 \times 0.1 = 0.01$. Then, the expected number of objects satisfying f is at least $0.01 \times O = 1 = k$.*

One search-minimal strategy to process f (see Section 3) searches on $e_1 \geq 0.2$, retrieving an expected 10 objects. Fagin's strategy is expected to retrieve a total of 20 objects (10 per subexpression).

Observe that although we can construct a search-minimal execution that accesses no more objects than Fagin's algorithm, the strategies in Section 3 are chosen based on their costs, not on the number of objects that they retrieve. Our optimality property (Theorem 3.10) guarantees that the cost of the chosen execution is not higher than that of the execution that retrieves no more objects than Fagin's plan.

If a repository is not uniquely graded, or if the statistics that we keep on the repository are not as finely grained as to allow for modeling the repository as uniquely graded, then Theorem 4.4 does not hold. However, Section 5 shows that the approach that we outlined in this section still is a desirable one when the assumptions of Theorem 4.4 do not hold strictly.

Although in this section we showed how to *process* a ranking expression like a filter condition, the semantics of both the filter condition and the ranking expression remain distinct. (See Section 2.) After processing a ranking expression as a filter condition, we have to compute the grade of the retrieved objects for the ranking expression, and sort them before returning them as the answer to the query.

Finally, note that when the query contains a filter condition F and a ranking expression R, it asks for k top objects by the ranking expression R that satisfy F. Using the results above, we can translate this query into the problem of optimizing the filter condition $F \wedge F'$, where F' is the filter condition associated with R assuming that we request $k' = \frac{k}{Sel(F)}$ top objects for R. We can then apply the methodology of Section 3.

5 Experimental Results

We ran a set of experiments using synthetic data. We assumed a database of 1,000,000 objects, with five attributes each. Attributes A_1 and A_2 are text attributes handled by vector-space search engines [5]. Attributes A_3 through A_5 are defined over images, and handled by R-trees indexes [6]. We use these attributes to build atomic expressions $e_i = Grade(A_i, v_i)$, for fixed values v_i, $i = 1, \ldots, 5$. Below we define the selectivities and the search and probe costs associated with the atomic expressions. (See [2] for a detailed description of the cost calculations and parameters.)

Expressions e_1 and e_2 (text attributes): We define $Sel(e_1, 1.0) = 3 \times 10^{-6}$ and $Sel(e_2, 1.0) = 7 \times 10^{-6}$, and assume that $Sel(e_i, g)$ increases exponentially as g decreases, up to $Sel(e_i, 0.0) = 1.0$. To compute the search cost for e_i and a grade g, we assume that the text engine that handles attribute A_i has an inverted file for A_i. To compute the probing cost for e_1, we assume that there is a hash table that, given an object id o, returns the weight vector associated with o. To compute the probing cost for e_2, we assume that there is no hash table for probing attribute A_2, and so a probe is processed as a search with the lowest grade.

Expressions e_3, e_4, and e_5 (image attributes): We assume that these attributes are managed using R trees, and use the methodology of [16] to estimate their parameters. The *fractal dimension* of a data set is a number that characterizes the distribution of the data. Given a range query, the fractal dimension associated with the data allows for accurate estimates of the query

$Sel(e_2, 1.0)$	% Improved Queries	% Improvement
1×10^{-6}	87.65	27.62
2×10^{-6}	89.30	28.35
3×10^{-6}	94.24	33.65

Table 2: Comparison of the optimal search-minimal executions against executions that choose the search condition without considering the probing cost.

selectivity and the number of nodes of the R tree that are accessed when processing the query. For our experiments, we set the dimension of the corresponding feature vectors of attributes A_3, A_4, and A_5 to be 9, 7, and 10, respectively, and their fractal dimension to be 9, 3, and 4, respectively. We then use the methodology in [16]. To compute the probing costs, we assume that A_3 and A_4 scan the entire database to look for the feature vector of the object being probed, whereas A_5 has a hash table for this task, like A_1.

5.1 Search-minimal Executions

In this section we report experimental results on optimizing strategies for a simple conjunctive filter condition $a_1 \wedge \ldots \wedge a_5$, where a_i is an atomic filter condition involving expression e_i, $i = 1, \ldots, 5$.

We compare the cost of the best search-minimal execution against the cost of the execution of the following strategy, which we call *Sep*. Strategy *Sep* is determined by first choosing the best atomic condition on which to search, considering the search cost and the selectivity of the conditions, but not the probe costs. Then, *Sep* probes the remaining conditions in an optimal order. Our search-minimal strategy differs from *Sep* in that we consider the probing costs to pick the search condition. We choose the grade for each atomic condition from the set $\{0.7, 0.8, 0.9\}$ to obtain $3^5 = 243$ different queries. Table 2 compares the cost of an optimal search-minimal execution versus the cost of the *Sep* strategy, and shows that even for the simple query form that we considered, we obtained improvements in the execution time of an overwhelming majority of the queries. For example, when $Sel(e_2, 1.0) = 3 \times 10^{-6}$, the search-minimal strategy outperformed *Sep* for 94.24% of the 243 queries. The *Sep* strategy was on average 33.65% more expensive for these queries. These results show the importance of considering the probe costs as well as the search costs when processing filter conditions.

5.2 Ranking Expressions as Filter Conditions

Section 4 showed how to map the execution of a ranking expression R into the execution of a filter condition F. If a repository is uniquely graded, there is a search-minimal execution for F that is expected to access no more objects than Fagin's algorithm for processing R directly (Theorem 4.4).

However, in practice, repositories might not be uniquely graded. And even if they are, the statistics that we keep to determine the selectivity of the atomic expressions for the different grades might not be "fine" enough for the theorem to hold.

For example, the statistics that we keep about an attribute might provide the selectivity of atomic conditions for that attribute at discrete grade points (e.g., at grades 0, 0.1, 0.2, ..., 1). In this section we consider such a case, and that the grade granularity for selectivity estimates is either 0.1 or 0.01.

The experiments in this section use two different ranking expressions over the set of five atomic expressions that we defined above: $R_{Min} = Min(e_1, e_2, e_3, e_4, e_5)$ and $R_{Max} = Max(e_1, e_2, e_3, e_4, e_5)$.

Figure 1 shows, for each k and for R_{Min}, the expected number of objects retrieved when we process the ranking expression as a filter condition, as a fraction of the number of objects that Fagin's algorithm is expected to retrieve, for two different statistics granularities. Even when we have only coarse statistics (grade granularity=0.1), the filter-condition strategy accesses less than one third as many objects as Fagin's algorithm. This fraction gets even lower for finer statistics (grade granularity=0.01). The reason is that the filter-condition strategy searches objects from only one atomic condition, whereas Fagin's algorithm retrieves objects from all five query conditions.

The results for the R_{Max} ranking expression (Figure 2) are not as good as for R_{Min}: both strategies retrieve objects from the five atomic conditions. Actually, when the grade granularity is 0.1, the filter-condition strategy accesses slightly more objects than Fagin's algorithm. This phenomenon disappears when the grade granularity is 0.01 and $k > 2$. For example, for $k = 10$, the filter-condition strategy accesses only an expected 30% of the objects that Fagin's algorithm is expected to retrieve.

6 Related Work

The concept of a graded match has been used extensively. For example, the query model in [3] allows specifying a grade of match as well as ranking. However, the processing of queries in [3] is based on searches (i.e., no probes are considered).

Many database systems support processing user-defined functions [17]. The QBIC system [4] from IBM Almaden allows users to query image repositories using image attributes like color, texture, and shapes. Another example is Cypress[2], a picture retrieval system that allows a filter condition to be specified, and returns a set of objects as the answer to the filter condition. However, Cypress does not support ranking. The querying interface supports user-defined functions and

[2] Accessible at `http://www.elib.berkeley.edu/cypress.html`.

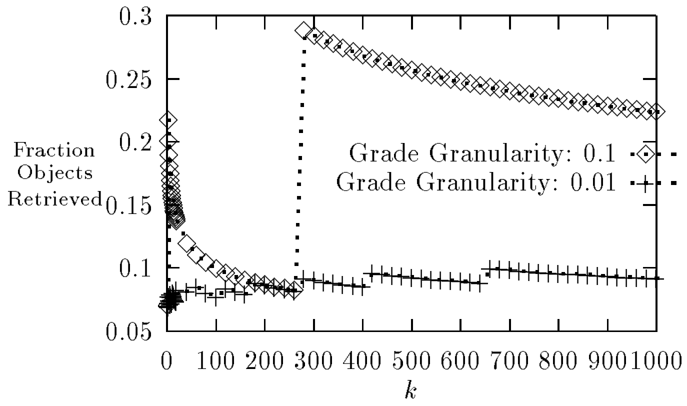

Figure 1: The expected number of objects retrieved when processing R_{Min} as a filter condition, as a fraction of the expected number for Fagin's algorithm, and as a function of the number of objects desired k.

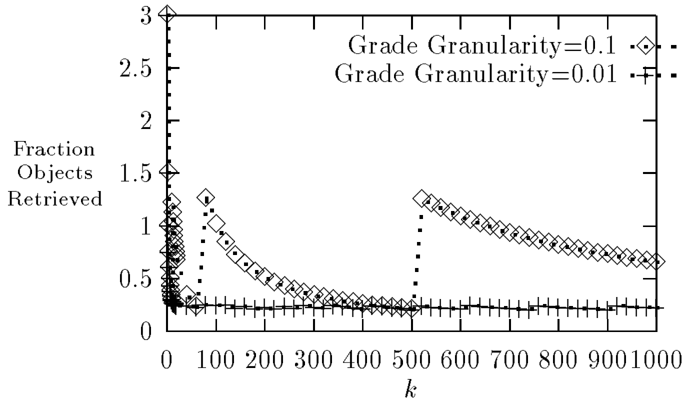

Figure 2: The expected number of objects retrieved when processing R_{Max} as a filter condition, as a fraction of the expected number for Fagin's algorithm, and as a function of the number of objects desired k.

predicates including a set of predefined graded matches (e.g., a predicate "mostly yellow").

The problem of optimizing user-defined filter conditions such as those in Cypress has been addressed in the literature. Work in [12, 13, 14] focuses on conjunctive selection conditions. Techniques to optimize arbitrary boolean selection conditions have been studied in [11, 18, 19]. Our work draws upon the known results in this area. (See Section 3.) However, all of the above work focuses on what we have referred to as *probing* costs, and does not consider the search costs.

On the other hand, the problem of determining an optimal set of conditions to search arises naturally when optimizing single-table queries with multiple indexes [20, 21]. The problem of sequencing the order of accesses to subfiles of transposed files is also closely related [22]. However, in the above contexts, the probing cost is either zero or is independent of the predicates. Our approach to defining the execution space is similar in spirit to [21], but our problem is more complex since probe costs can be significant as well as varied. When the filter condition is restricted to being a conjunction, the optimization problem can be cast as a join-ordering problem [10, 13, 14]. However, such a formulation fails to capture characteristics that are particular of selection queries. In summary, past work in this area does not consider the case where the search cost as well as the probing cost need to be considered for optimization of arbitrarily complex filter conditions containing and's and or's.

In the context of the Garlic project at IBM Almaden [23], Fagin's recent work [1] focuses on how to evaluate queries that ask for a few top matches for a ranking expression. (See Section 4.1.) Our ranking expressions are a special case of Fagin's queries. Under broad assumptions on the cost model, Fagin demonstrates the optimality of his algorithm for a class of composition functions. Our contribution has been to show that ranking expressions can be evaluated efficiently by using *GradeSearch*, and that the expected number of retrievals of objects in this technique is provably no larger than in Fagin's approach. This result clears the way for the integration of the evaluation of filter conditions and ranking expressions.

Acknowledgments

We thank Umesh Dayal, Héctor García-Molina, Jeff Ullman, Tak Yan, and the entire database group at HPLabs for helpful discussions and comments. We also thank Ron Fagin for sending us an early draft of [1].

References

[1] Ronald Fagin. Combining fuzzy information from multiple systems. In 15^{th} *ACM Symposium on Principles of Database Systems*, June 1996. Also available as IBM Almaden Research Center Technical Report RJ 9980.

[2] Surajit Chaudhuri and Luis Gravano. Optimizing queries over multimedia repositories. Technical report, Hewlett-Packard Laboratories, March 1996. Also available as `ftp://db.stanford.edu/pub/gravano/-1996/sigmod.ps`.

[3] F. Rabitti. Retrieval of multimedia documents by imprecise query specification. In *Proceedings of the 1990 EDBT*, Venice, Italy, March 1990.

[4] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, and C. Faloutsos. The QBIC project: Querying images by content using color, texture, and shape. In *Storage and retrieval for image and video databases (SPIE)*, pages 173–187, February 1993.

[5] Gerard Salton. *Automatic text processing: the transformation, analysis, and retrieval of information by computer.* Addison Wesley, 1989.

[6] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD Conference*, pages 47–57, June 1984.

[7] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R* tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD Conference*, pages 322–331, May 1990.

[8] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th Conference on Very Large Databases*, pages 507–518, September 1987.

[9] Nick Roussopoulos, Stephen Kelley, and Frederick Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD Conference on the Management of Data*, pages 71–79, San Jose, CA, May 1995.

[10] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Symposium on Management of Data*, pages 23–34, Boston, MA, June 1979.

[11] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing boolean expressions in object bases. In *Proceedings of the 18th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Vancouver*, August 1992.

[12] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the ACM SIGMOD International Symposium on Management of Data*, Washington D. C., May 1993.

[13] T. Ibaraki and T. Kameda. On the optimal nesting order for computing N-relational joins. *ACM Transactions on Database Systems, ; ACM CR 8506 0535*, 9(3), September 1984.

[14] R. Krishnamurthy, H. Boral, and C. Zanialo. Optimization of nonrecursive queries. In *Proceedings of International Conference on Very Large Data Bases*, pages 128–137, Kyoto, Japan, August 1986.

[15] Luis Gravano and Héctor García-Molina. Generalizing GlOSS for vector-space databases and broker hierarchies. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 78–89, September 1995.

[16] Christos Faloutsos and Ibrahim Kamel. Beyond uniformity and independence: analysis of R trees using the concept of fractal dimension. In 13^{th} *ACM Symposium on Principles of Database Systems*, May 1994.

[17] M. Carey and L. Haas. Extensible database management systems. *ACM SIGMOD Record*, December 1990.

[18] A. Kemper, G. Moerkotte, and K. Peithner. Optimizing disjunctive queries with expensive predicates. *SIGMOD record*, 23(2):336, 1994.

[19] M.T. Ozsu and D. Meechan. Finding heuristics for processing selection queries in relational database systems. *Information Systems*, 15(3), 1990.

[20] A. Rosenthal and D. Reiner. An architecture for query optimization. In *Proc. ACM SIGMOD Conf.*, page 246, Orlando, FL, June 1982.

[21] C. Mohan. Single table access using multiple indexes: Optimization, execution and concurrency control techniques. In *EDBT 90, Venice*, 1990. Also published in/as: IBM Almaden Res.Ctr, Res.R. No.RJ7341, Mar.1990, 15pp.

[22] Don S. Batory. On searching transposed files. *ACM Transactions on Database Systems*, 4(4), December 1979.

[23] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers. Towards heterogeneous multimedia information systems: the Garlic Approach. In *RIDE-DOM 1995*, Taipei, Taiwan, 1995.