Technical Report CUCS-017-02
Columbia University, August 2002

Optimizing Top-K Selection Queries over Multimedia Repositories*

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Luis Gravano
Columbia University
gravano@cs.columbia.edu

Amélie Marian
Columbia University
amelie@cs.columbia.edu

Abstract

Repositories of multimedia objects having multiple types of attributes (e.g., image, text) are becoming increasingly common. A query on these attributes will typically request not just a set of objects, as in the traditional relational query model (*filtering*), but also a *grade of match* associated with each object, which indicates how well the object matches the selection condition (*ranking*). Furthermore, unlike in the relational model, users may just want the k top-ranked objects for their selection queries, for a relatively small k. In addition to the differences in the query model, another peculiarity of multimedia repositories is that they may allow access to the attributes of each object only through indexes. In this paper, we investigate how to optimize the processing of top-k selection queries over multimedia repositories. The access characteristics of the repositories and the above query model lead to novel issues in query optimization. In particular, the choice of the indexes used to search the repository strongly influences the cost of processing the filtering condition. 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 present an efficient algorithm that solves the problem optimally when the predicates in the query are independent. We also show that the problem of optimizing top-k selection queries can be viewed, in many cases, as that of evaluating more traditional selection conditions. Thus, both problems can be viewed together as an extended filtering problem to which techniques of query processing and optimization may be adapted.

*Work done in part while the authors were at Hewlett-Packard Laboratories.

1 Introduction

The problem of content management of multimedia repositories is becoming increasingly important with the development of multimedia applications and the web [21]. 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. To access such large repositories efficiently, we need to store information on attributes of the multimedia objects. Such attributes include the date the multimedia object was authored, a free-text description of the object, and 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 [14, 15, 16]. Thus, an atomic filter condition will not be an equality between two values (e.g., between a given color c_0 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_0*)(oid)* > 0.7). Next, every condition on an attribute of a multimedia object may only be evaluated through calls to a system or index that handles that particular attribute. 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[1]. 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). For example, consider a filter condition consisting of a conjunction of two atomic conditions. If we search on the first condition and probe on the second, the latter benefits from the reduction in the number of objects that need probing, due to the selectivity of the

[1] The queries identify a candidate set (or list) of objects for displaying. How to actually display these objects is an important problem that we do not address in this paper.

first condition.

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 constant) has been studied before. When the filter condition is a conjunction of atomic conditions, the problem becomes closely related to that of ordering joins. 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.

To optimize the processing of a filter condition, we define a space of *search-minimal executions*, and show an optimal strategy in that space for the case when the conditions present in the filter condition are *independent*. Although the search-minimal execution space is a restricted space, our experiments indicate that if we introduce a simple post-optimization step for conjunctive conditions, we obtain plans that are nearly always as efficient as the plans obtained when plans are not restricted to be search minimal. Our experiments also show that considering both the search and probe costs during query optimization impacts the choice of an execution plan significantly. Also, we prove that if the conditions in the filter condition are not independent, the problem of determining an optimal search-minimal execution is NP-hard.

Our paper also contributes to the problem of optimizing the evaluation of queries that contain ranking expression. Previous significant work in this area is due to Fagin [14, 15, 16], who shows his approach 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 efficiently. Our experimental results indicate that such processing of ranking expressions as filter conditions is often quite efficient. Unlike Fagin's work, our optimization and evaluation technique is heuristic (as in relational query optimization). However, from a practical systems perspective, our technique is of significance since for the first time it provides an ability to treat queries that contain both filter and ranking expressions in a uniform framework for query optimization and evaluation with few extensions to core query processing techniques.

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. Section 6 is devoted to related work. We conclude with a summary and a few interesting questions for future work in Section 7.

2 Query Model

In this section we introduce a query model to *select* multimedia objects from a repository. (See [32] 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. *Exact* matches such as *o.attr = value* can be represented by the filter condition *Grade(attr,value)(o)* $\geq$ *1*. However, in this paper, we will not discuss how exact matches can be treated especially.

Following [14, 15, 16], 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*. (Fagin's expressions are more general in that he allows other composition functions.) Every object has a grade between 0 and 1 for a given ranking expression. Users can then use a ranking expression in their queries, and ask for k objects with the top grades for the given ranking expression. In this paper, we assume that ties are broken arbitrarily. An alternative semantics, which we do not pursue in this paper, is that if there are ties, all objects with the same grade are returned, even if that exceeds the required number of objects k.

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. Intuitively, 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 (profile), a scanned fingerprint (fingerprint), and a recording of a voice sample (voice_sample). Given a fingerprint F and a voice sample V, the following example asks for records whose fingerprint matches F well. Alternatively, a record is also acceptable if its profile matches the string 'on parole' with grade 0.9 or higher, and its 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(voice_sample, V) >= .5 AND Grade(profile, 'on parole') >= .9)
       OR (Grade(fingerprint, F) >= .9)
ORDER [10] BY Max(Grade(fingerprint, F), Grade(voice_sample, 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 object. We use these grades for ordering the objects that satisfy the filter condition.

Given a filter condition F and a ranking expression R, 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.)

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.

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. (If k' objects satisfy F in db, and $k' < k$, then use k' instead of k below.)

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 necessarily 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. We will show that any of the four possibilities for R_F produces a wrong answer for the query:*

- *Case $R_F = e_1$, or $R_F =$ Min(e_1, e_2): The top object for R_F is o_3, which is a wrong answer.*

- *Case $R_F = e_2$, or $R_F =$ Max(e_1, e_2): The top object for R_F is o_1, which is a wrong answer.*

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 closely enough, as we will 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 we assume multimedia repositories support. (See for example [29].) Key to these interfaces is that the objects match attribute values with a grade of match, as we discussed above.

- *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 a *Probe* call atop *GradeSearch* by requesting all objects that match a given attribute value with at least some specified grade, and then decreasing this grade until the grade for the object requested in the *Probe* call is obtained. A similar strategy could be implemented atop *TopSearch*. 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 [36, 1]. In such a model, the value of an object for attribute T is regarded as a traditional document. Then, given a query value for attribute 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. To compute these similarities, vector-space retrieval systems typically represent both documents and queries as weight vectors, where each weight corresponds to a term in the vocabulary. Given a query, a vector-space retrieval system returns a list of the matching documents sorted by their grade for the query. The grade –or similarity– of a document and a query is usually computed by taking the inner product of their weight vectors. 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*, as discussed above.

Image Attributes:

Other popular attributes are features of images. If the objects of a repository contain an image, an attribute of the objects could be the color histogram of this image. Then, a filter condition on such an attribute can ask for objects whose image histogram matches a given color histogram closely, for example. The QBIC system supports this type of queries [29]. One of the most popular ways of handling such attributes and queries is by using R trees [22] and its variants [3, 38] to index the feature vectors associated with the attributes. The grade between two feature vectors is computed based on the semantics of the attributes, and sophisticated algorithms have been developed in the context of the QBIC project, for instance [18].

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. [34] has 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 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 since the System-R optimization effort [37].

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

1. *Every atomic filter condition occurs at most once in f.*

2. *Every n atomic filter conditions $a_1, \ldots, a_n$*[2] *satisfy the following: $p(a_1 \wedge \ldots \wedge a_n) = \Pi_{i=1}^{n} p(a_i)$, where $p(a)$ is the probability that the filter condition a is true.*

Independence rules out filter conditions with repeated attributes, and also filter conditions with, for example, two atomic conditions a_1 and a_2 such that a_1 is true (or false) whenever a_2 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, and check the remaining conditions on these objects through *Probe* operations.

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 processing of expensive filter conditions [26, 23, 24, 27, 11].

In this section, we first define a space of *search-minimal* executions, which access as few attributes as possible using *GradeSearch*, and sketch the cost model and the optimization criteria. Next, we describe an optimization algorithm and explain the conditions under which it is optimal. Finally, we show how we can further improve the execution plan produced by our algorithm through a simple "post-optimization" step to lower the cost of the original plan, and 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, which show that considering both the search and probe costs leads to significantly better execution strategies, and that post-optimized search-minimal executions behave almost as well as the best (not necessarily search-minimal) executions.

[2]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*.

3.1 Execution Space

As an introduction, 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 $a_1 \vee \ldots \vee a_n$. 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. *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 search-minimal strategies represent a subset of the possible executions. In particular for a conjunctive filter condition, instead of searching on a single subcondition 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 particular, there are exponentially many subsets of conjuncts to search on, but only a linear number of minimal conjunct sets for searching.

Intuitively, a search-minimal execution evaluates a minimal set of atomic conditions using *GradeSearch*, and evaluates the rest of the conditions using *Probe*. A simple conjunctive filter condition needs to use

GradeSearch for only one atomic condition. However, an arbitrary filter condition involving $\wedge$'s and $\vee$'s might need to search more than one atomic condition, like the disjunction above.

We are motivated by several factors to focus on search-minimal executions. First, as discussed in the context of conjunctive queries, search-minimal executions avoid an explosion in the search space. Next, as we will discuss in Section 3.4 as well as demonstrate experimentally in Section 5, simple post-optimizations allow us to derive from the optimal search-minimal execution a cheaper execution that is not necessarily search-minimal.

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 entire filter condition. 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 in which the internal nodes correspond to the boolean connectives (hence there are "$\wedge$ nodes" and "$\vee$ nodes") and the leaf nodes correspond to the atomic conditions in f. Let a be 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:*

$$f = a_4 \wedge ((a_1 \wedge a_2) \vee a_3)$$

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.

Proposition 3.4: *Let f be an independent filter condition, and a be an atomic condition of f. Then $a \wedge R(a, f) \Rightarrow f$.*

Proof: By induction on the structure of the filter condition f. If $f = a$, then $R(a, f)$ = true. Thus the proposition follows trivially.

Now, consider the case $f = f_1 \wedge \ldots \wedge f_n$. Assume that a appears in f_1 (and nowhere else, because f is independent). From the definition of residue, $R(a, f) = R(a, f_1) \wedge f_2 \wedge \ldots \wedge f_n$. From the inductive hypothesis, $a \wedge R(a, f_1) \Rightarrow f_1$. Then, $a \wedge R(a, f) = a \wedge R(a, f_1) \wedge f_2 \wedge \ldots \wedge f_n \Rightarrow f_1 \wedge f_2 \wedge \ldots \wedge f_n = f$.

Next, consider the case $f = f_1 \vee \ldots \vee f_n$. Assume that a appears in f_1. From the definition of residue, $R(a, f) = R(a, f_1)$. From the inductive hypothesis, $a \wedge R(a, f_1) \Rightarrow f_1$. Then, $a \wedge R(a, f) = a \wedge R(a, f_1) \Rightarrow f_1 \vee \ldots \vee f_n = f$. ∎

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 3.5: *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 3.6: *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 3.7: *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.

Proof:

- $\bigvee_{a \in m}(a \wedge R(a, f)) \Rightarrow f$: Follows directly from Proposition 3.4 and from the fact that every condition has at least one atomic condition.

- $f \Rightarrow \bigvee_{a \in m}(a \wedge R(a, f))$: By induction on the structure of f. If $f = a$, then the results follows directly.

  Now consider the case $f = f_1 \wedge \ldots \wedge f_n$. Because m is complete for f, there must exist $m_i \subseteq m$ such that m_i is a complete set of atomic conditions for f_i, for some $1 \leq i \leq n$. Since m_i is complete for f_i, and using the inductive hypothesis, it follows that $f_i \Rightarrow \bigvee_{a \in m_i}(a \wedge R(a, f_i))$. Then, $f = f_1 \wedge \ldots \wedge f_n \Rightarrow \bigvee_{a \in m}(a \wedge R(a, f_i) \wedge f_1 \wedge \ldots f_{i-1} \wedge f_{i+1} \wedge \ldots \wedge f_n) = \bigvee_{a \in m}(a \wedge R(a, f))$.

  Finally, consider the case $f = f_1 \vee \ldots \vee f_n$. Because m is complete for f, there exists $m_i \subseteq m$ such that m_i is a complete set of atomic conditions for f_i, for all $i = 1, \ldots, n$. From the inductive hypothesis, $f_i \Rightarrow \bigvee_{a \in m_i}(a \wedge R(a, f_i))$. Then, $f_1 \vee \ldots \vee f_n \Rightarrow \bigvee_{a \in m}(a \wedge R(a, f))$, because $R(a, f_i) = R(a, f)$, for all $i = 1, \ldots, n$. ▮

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

Definition 3.8: *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. We then show how to further optimize these plans to lower their cost (Section 3.4). The strategies that result from these post-optimizations are not search-minimal executions.

3.2 Assumptions and 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.

Statistical Parameters:

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

- *Selectivity Factor Sel*(a)*:* Fraction of objects in the repository that satisfy condition a.
- *Search Cost* $SC(a)$*:* Cost of retrieving the ids of the objects that satisfy condition a using *GradeSearch*.
- *Probe Cost* $PC(a, p)$*:* Cost of checking 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 will be the key of the post-optimization step of Section 3.4.

We now sketch how to estimate these parameters over multimedia repositories for text and image attributes. Consider first a textual attribute that is handled by a vector-space retrieval system. Typically such a system has inverted lists associated with each term in the vocabulary [36, 1]. 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 [20] 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 [19], 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.

Assumptions on Conditions:

As we mentioned before, we will restrict our discussion to optimizing independent filter conditions. We can compute the selectivities of complex independent filter conditions using the following two rules as in traditional optimization [37]:

- *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. Subsequently, 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 Search Minimal Executions:

To pick the least expensive search-minimal execution, we need to define the cost of such executions. As we can see from the definition of the search-minimal executions, the cost of one such execution depends on (a) the choice of the search conditions, (b) the probing costs of the remaining conditions, and (c) the cost of taking the union of the answer sets. Value (c) dominates only when the selectivity of the filter condition is low. Therefore, to simplify the optimization problem, we focus only on the search and probe costs.

Given a search-minimal condition set m for a filter condition f and an algorithm w for probing conditions, we now 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, $|O_a| = \mathit{Sel}(a) \cdot O$.

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 [26]. (See also [24, 27, 23, 11].)

If $R(a, f)$ is a conjunction of atomic conditions $a_1 \wedge \ldots \wedge a_n$, there is an efficient algorithm w that finds the optimum probing strategy. Specifically, it can be shown [24, 27] that the order in which the atomic conditions for each object should be probed is given by the *rank* of each condition a_i, defined as $\frac{Sel(a_i)-1}{c_i}$ if we assume that $PC(a_i, p) = c_i \cdot p$ for some constant c_i and where p is the number of objects to probe. Then, we can calculate the cost $PC(R(a, f), p)$ as follows, assuming for simplicity 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 $S_i = Sel(a_1) \cdot \ldots \cdot Sel(a_{i-1}) \cdot p \cdot c_i$. (Note that $Sel(a_1) \cdot \ldots \cdot Sel(a_{i-1}) \cdot p$ is the number of objects that satisfy conditions $a_1, \ldots, a_{i-1}$ out of the original p objects; we only need to probe these objects for condition a_i, at a cost of c_i for each object.) This result is well known and was observed in the database context by [24, 27]. We can take a similar approach to order the evaluation of a disjunction of atomic conditions.

Example 3.9: *Consider the filter condition $a_1 \wedge a_2 \wedge a_3$, where $Sel(a_1) = .01$, $Sel(a_2) = .02$, and $Sel(a_3) = .05$. Let $c_1 = c_2 = 1$, and $c_3 = .5$. The increasing rank sequence is then c_3, c_1, c_2. Then, the probing cost for 1000 objects is as follows:*

$$(.5 + .05 \cdot 1 + .05 \cdot .01 \cdot 1) \cdot 1000 = 550.5$$

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 [26]. Therefore, we assume that we exploit one of these available techniques 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:

Given that we can compute the cost metric $C(f, m)$ for any independent filter condition f and condition set m, our goal is to pick an optimal search-minimal condition set. Let $M(f)$ be the set of all search-minimal condition sets for f.

Definition 3.10: *A search-minimal condition set m for an independent filter condition f is* optimal *if $C(f, m) = \min_{m' \in M(f)} C(f, m')$*

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

Definition 3.11: *Let f be a filter condition and f' be a subcondition 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 3.12: *Let f be an independent filter condition. Then $SM_f(f)$ is an optimal search-minimal condition set for f.*

Before we can prove Theorem 3.12, we need the following auxiliary result.

Proposition 3.13: *Let f_1 and f_2 be two independent filter conditions with no atomic conditions in common. Then:*

1. $M(f_1 \wedge f_2) = M(f_1) \cup M(f_2)$
2. $M(f_1 \vee f_2) = M(f_1) \odot M(f_2)$, *where $m \in M(f_1) \odot M(f_2)$ if and only if $\exists m_1 \in M(f_1)$, $m_2 \in M(f_2)$ such that $m = m_1 \cup m_2$*

Proof:

We will first show that $M(f_1 \wedge f_2) = M(f_1) \cup M(f_2)$ (part 1 of the proposition):

- $M(f_1 \wedge f_2) \subseteq M(f_1) \cup M(f_2)$.

  Consider $m \in M(f_1 \wedge f_2)$. Then, either $\exists m_1$ complete for f_1 such that $m_1 \subseteq m$, or $\exists m_2$ complete for f_2 such that $m_2 \subseteq m$. (Otherwise, m would not be complete for $f_1 \wedge f_2$.) Assume that $\exists m_1$ complete for f_1 such that $m_1 \subseteq m$. Any object that satisfies $f_1 \wedge f_2$ also satisfies f_1, and at least one condition of m_1, because m_1 is complete for f_1. Then, m_1 is also complete for $f_1 \wedge f_2$. But $m_1 \subseteq m$, and $m \in M(f_1 \wedge f_2)$. Therefore, $m = m_1 \in M(f_1)$.

- $M(f_1) \cup M(f_2) \subseteq M(f_1 \wedge f_2)$.

  Consider $m \in M(f_1) \cup M(f_2)$. Furthermore, suppose that $m \in M(f_1)$. It is easy to see that m is complete for $f_1 \wedge f_2$. To see that m is also search-minimal for $f_1 \wedge f_2$, consider $m' \subset m$ that is also complete for $f_1 \wedge f_2$. Because m is search-minimal for f_1, it must be the case that m' is not complete for f_1. Then, there is an object o that satisfies f_1 and none of the conditions in m'. But then we can build a new object o' that also satisfies f_2, and still does not satisfy any of the conditions in m', because f_1 and f_2 do not share any conditions, and $m' \subset m \in M(f_1)$. However, o' would contradict the completeness of m' for $f_1 \wedge f_2$. Therefore, m is also search-minimal for $f_1 \wedge f_2$.

Now, we will show that $M(f_1 \vee f_2) = M(f_1) \odot M(f_2)$ (part 2 of the proposition).

- $M(f_1 \vee f_2) \subseteq M(f_1) \odot M(f_2)$:

  Consider $m \in M(f_1 \vee f_2)$. Let m_1 (resp., m_2) be the restriction of m to conditions in f_1 (resp., in f_2). It is easy to see that $m_1 \in M(f_1)$ and $m_2 \in M(f_2)$. Therefore, $m = m_1 \cup m_2 \in M(f_1) \odot M(f_2)$.

- $M(f_1) \odot M(f_2) \subseteq M(f_1 \vee f_2)$: Straightforward. ∎

Proof (Theorem 3.12): From Proposition 3.13 it is clear that $SM_f(f')$ is a search-minimal condition set for f'. We will use induction on the structure of f' to show that $\forall$ subcondition f' of f, $\forall m \in M(f'), C(f, SM_f(f')) \leq C(f, m)$.

- Case $f' = a$: Straightforward.

- Case $f' = f_1 \wedge \ldots \wedge f_n$: Let $m = SM_f(f')$. Suppose that $\exists m' \in M(f')$ such that $C(f, m') < C(f, m)$. From Proposition 3.13, $m' \in M(f_i)$, for some $1 \leq i \leq n$. From the inductive hypothesis, $C(f, m') \geq C(f, m_i)$, where $m_i = SM_f(f_i)$. And from construction of $SM_f(f')$, $C(f, m) \leq C(f, m_i)$. Therefore, $C(f, m') \geq C(f, m)$, contradicting our choice of m'.

- Case $f' = f_1 \vee \ldots \vee f_n$: Let $m = SM_f(f')$. Suppose that $\exists m' \in M(f')$ such that $C(f, m') < C(f, m)$. From Proposition 3.13, $m' = \cup_{i=1,\ldots,n} m'_i$, where $m'_i \in M(f_i)$. From the inductive hypothesis, $C(f, m'_i) \geq C(f, m_i)$, where $m_i = SM_f(f_i)$, $i = 1, \ldots, n$. Therefore, because f is independent and using the definition of SM_f, $C(f, m) = \sum_{i=1}^{n} C(f, m_i) \leq \sum_{i=1}^{n} C(f, m'_i) = C(f, m')$, contradicting our choice of m'. ∎

Our strategy requires that we compute the cost of each atomic condition at most once, since the cost and search-minimal set are computed "bottom-up."

The problem of determining an optimal evaluation strategy for a filter condition as discussed in this paper is related to the problem of choosing access paths for traditional selection queries in the presence of indexes for a query processor that supports index union and intersection [33, 28]. In this paper, we restrict ourselves to search-minimal executions but do allow for probe costs. Please see the related work section for additional details.

The proof of optimality of $SM_f(f)$ depends on the fact that the given filter condition f is independent. [3] Nonetheless, with the following simple modification, we can still provide a search-minimal condition set in case the given condition is not independent. However, this set is is no longer guaranteed to be optimal:

1. Derive $SM_f(f)$ assuming f is an independent condition and treating each occurrence of a condition as a new atomic condition.

2. Identify a subset $m \subseteq SM_f(f)$ that is search minimal for f.

Observe that the first step ensures completeness whereas the second step ensures that the set m is minimal and can be determined efficiently. However, as the following example shows, such a heuristic does not always result in an optimal search-minimal condition set.

Example 3.14 *Assume that the filter condition is* $(a \wedge b) \vee (a \wedge c)$. *The first step of the algorithm treats every instance of* a *as a different condition. So, the query is viewed by Step (1) as* $(a_1 \wedge b) \vee (a_2 \wedge c)$. *Assume that the algorithm determines* $SM_f(f) = \{b, c\}$. *Step (2) of the algorithm does not change* $SM_f(f)$, *although* $\{a\}$ *could be a significantly better search-minimal condition set. Therefore, the algorithm may fail to identify the best search-minimal condition set if the subconditions are not independent, as in this example.*

The above result is not surprising given that the general optimality problem, where no assumptions are made about independence, is intractable even for the very simple cost model where search cost is 1 and probe cost is 0, as the following theorem shows.

[3] Less expensive executions might be possible if independence does not hold and extra information is available during query planning. As a simple example, consider a filter condition $a_1 \vee a_2$, where a_2 is true every time that a_1 is true. In this case, there would be no need to search on a_2 to find all objects that match the whole condition.

Theorem 3.15: *The problem of determining an optimal search-minimal condition set for a filter condition is NP-hard.*

Proof: We prove the result by a reduction from the vertex-cover problem. To map an instance of the vertex-cover problem $G = (V, E)$ to our problem we generate a filter condition F such that G has a vertex cover of size k or less if and only if there is a processing strategy for F that retrieves objects using searches over k or fewer atomic conditions. We associate a unit cost for every search, and zero cost for the probes to complete the proof.

Given the (undirected) graph $G = (V, E)$, we generate the following filter condition:

$$F = \bigvee_{(v_i, v_j) \in E} (v_i \wedge v_j)$$

where the v_i's are atomic conditions. We define $PC(f, p) = 0$ for all f, p, and $SC(v_i) = 1$ for all the v_i's. Therefore, the cost of any search-minimal condition set m is the number of atomic conditions in m.

Now, G has a vertex cover of size k or less if and only if there is a search-minimal condition set for F with k conditions or less:

- $\Rightarrow$: Assume G has a vertex cover V' of size k or less. Then, there is a set of atomic conditions V' of size k or less such that for each subcondition $v_i \wedge v_j$ of F, either $v_i \in V'$ or $v_j \in V'$.

- $\Leftarrow$: Assume that there is a search-minimal condition set m for F with k or fewer conditions. Suppose that there is a subexpression $v_i \wedge v_j$ such that $v_i, v_j \notin m$. Then suppose that there is an object o that satisfies only atomic conditions v_i and v_j, and none of the others. Then o satisfies F, from the construction of F, but it does not satisfy any of the conditions in m, contradicting the completeness of m. Therefore, either v_i or v_j are in m. Consequently, m defines a vertex cover for G with k or fewer elements. ∎

3.4 Post-optimization: Beyond Picking a Search-Minimal Set

While choosing an optimal search-minimal condition set is a key step in selecting an efficient execution plan, there are several other opportunities for optimization.

First, we note that a search-minimal execution for a filter condition f always handles the residue of a search condition a by probing the condition $R(a, f)$. However, when the number of objects to be probed is high, the cost of probing $R(a, f)$ may exceed the cost of searching on the atomic condition a using *GradeSearch*. Thus, in case of a conjunctive query, it may be more efficient to use more than one condition for searching. In other words, it could be convenient to allow the conditions that are used for searching to no longer form a search-minimal condition set. However, our optimization algorithm does not consider such a plan.

To address this lack of flexibility, we introduce a post-optimization step that locally replaces probes on one or more conditions by the corresponding searches, as the following example illustrates.

Example 3.16: *Assume that the optimal search-minimal execution for $a_1 \wedge a_2 \wedge a_3$ searches on condition a_1, and probes on conditions a_2 and a_3. Let the number of objects probed by a_2 be 1000, and the probe cost be 1 unit for every probe. Thus, the total cost of probing on a_2 is 1000 units. If the search cost on a_2 is 800, then we can modify the execution plan a posteriori to search on conditions a_1 and a_2, and to probe just on a_3.*

In Section 5 we report results on an experimental evaluation of this simple post-optimization step.

In addition to turning certain probes into searches, our algorithm presents other less critical opportunities for post-optimization. For example, when processing several atomic conditions we could also improve how we "merge" the objects retrieved using each of these conditions: (1) An object that is retrieved by searches on both a_1 and a_2, can be probed using either the residue $R(a_1, f)$ or the residue $R(a_2, f)$. Such a choice can be cost-based and influences the order in which we merge results from multiple searches. (2) The merging order is also influenced by the cost of detecting and eliminating duplicate objects, and by the size of the answer sets resulting from searches. Evaluating alternatives for this "merging" and determining their effect on the execution costs remains as future work.

4 Filter Conditions and Ranking Expressions

In this section, we consider queries each of which consists not only of a filter condition, but also of a ranking expression. The answer to such queries consists of the top objects for the ranking expression that also satisfy the filter condition. We first look at queries consisting only of ranking expressions (Section 4.1). Section 4.2 describes an algorithm for processing this type of queries that has been presented in references [14, 15, 16]. Finally, Section 4.3 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 mapping is central to processing queries with ranking expressions applying the techniques of Section 3 for processing filter conditions. The experimental results of Section 5 show that, in some cases, the number of objects retrieved and probed when processing a ranking expression as a filter condition can be considerably smaller than when processing the ranking expression using the algorithm in [14, 15, 16].

4.1 Ranking Expressions

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.

Example 4.1: *Consider the ranking expression* e_1 *=Max(Grade(fingerprint, F), Grade(profile, P)). Expression* e_1 *favors objects with either fingerprints matching the given value* F *closely, or with text profiles matching the given profile* P *closely. Alternatively, consider the ranking expression* e_2 *=Min(Grade(fingerprint, F), Grade(profile, P)). Expression* e_2 *favors objects with good matches for both their fingerprints and profiles.*

4.2 Fagin's Strategy

Fagin presented a novel approach to processing a query consisting of a ranking expression in references [14, 15, 16]. In this section we briefly describe his approach. In Section 5, we experimentally compare this algorithm against our approach for processing ranking expressions using a modified version of our techniques of Section 3.

Consider a ranking expression $R = \mathit{Min}(a_1, \ldots, a_n)$, 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. It does so by retrieving the top objects from each of the subexpressions 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 per subexpression of R.) Fagin proved that the set of objects 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 the above algorithm to retrieve k of the best objects for an expression R that is a *Min* of independent atomic expressions is asymptotically optimal with arbitrarily high probability.

Now, consider a ranking expression $R = \mathit{Max}(a_1, \ldots, a_n)$, where the a_i's are independent atomic expressions. Suppose that we are interested in k objects with the highest grades for R. In this case, another algorithm by Fagin requests exactly k objects from each of the subexpressions a_i, $i = 1, \ldots, n$, with no need to probe any objects. It follows that there are k top objects for R among these $k \cdot n$ objects.

4.3 Processing Ranking Expressions as Filter Conditions

As discussed in Section 2, a query may have both a filter condition as well as a ranking expression. A naive query-execution strategy might stage the processing of these two components of the query, leading to two alternatives: (a) evaluate the filter condition first using the techniques in Section 3.3, and then rank the results by probing on the necessary attributes; (b) use techniques for efficient top-k query processing to identify the top-k' objects for the ranking expression (for some $k' \geq k$), and then filter out any objects that do not satisfy the filter condition by probing on the necessary attributes. Note that the second strategy requires deriving a value of k' from the given k by somehow taking into account the selectivity of the filter condition. Both of these alternatives ignore the possible synergy in optimizing the execution of queries by considering filter conditions and ranking expressions *simultaneously*. Our goal in this section is to precisely identify if we could view filtering and ranking in a uniform framework. This brings up the challenge of "mapping" ranking expressions into a filter condition without significant loss of efficiency as compared to using techniques that are optimized for ranking expressions. However, since such mapping takes place as part of query optimization, we must depend on estimation techniques to derive a suitable filter condition. We do so by techniques similar to those adopted for cost estimation in traditional relational databases. Thus, our "mapped" ranking expressions are optimized not in an absolute sense but leveraging approximate statistics that are available. This is in sharp contrast to the techniques by Fagin [14, 15, 16] that we outlined above, which provide theoretical performance guarantees. However, our mapping technique has the benefit of enabling a smooth integration with the broader class of queries involving filter conditions. Moreover, as our experiments will suggest, our technique compares favorably to Fagin's.

Although in this section we show how to *process* a ranking expression as a filter condition, the semantics of both the filter condition and the ranking expression remain distinct. (See Section 2.) We still need to specify a ranking expression to get a sorted set of objects. Filter conditions have unordered sets as their answers. In the strategy that we describe below, after processing a ranking expression as a filter condition, we will 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.

Given a ranking expression R and the number k of objects desired, we give an algorithm to assign a grade to each atomic expression in R, and a filter condition F with the same "structure" as R that retrieves the top objects according to R.

Example 4.2: *Consider a ranking expression:*

$$e = \textit{Min}(\textit{Grade}(A_1, v_1), \textit{Grade}(A_2, v_2))$$

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 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. Furthermore, such a grade G should be as high as possible, to retrieve as few objects as possible.

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:*

$$f = (\textit{Grade}(A_1, v_1)(o) \geq G) \wedge (\textit{Grade}(A_2, v_2)(o) \geq G)$$

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.

Now, consider a ranking expression:

$$e' = \textit{Max}(\textit{Grade}(A_1, v_1), \textit{Grade}(A_2, v_2))$$

where A_i and v_i are as before, $i = 1, 2$. We want two objects with the top grades for e'. If we somehow find a grade G' such that there are at least two objects with grade G' or higher for expression e', we can process e' by processing the following filter condition f':

$$f' = (Grade(A_1, v_1)(o) \geq G') \vee (Grade(A_2, v_2)(o) \geq G')$$

By processing f' we retrieve all of the objects having grade G' or higher for the ranking expression e', with no need to probe any objects. If there are at least two such objects, then we can answer e' by returning two objects with the top grades for e' from among the set of objects that we retrieved.

As we have seen in the example above, we can process a ranking expression e as a filter condition f followed by a sorting step. The key point in the mapping of the problem from a ranking problem to a (modified) filtering problem lies in finding the grade G to use in f, as in the example. Ideally, grade G should be the k^{th} largest grade of any object in the database: the resulting filter condition f that uses such a value of G would then retrieve exactly the top objects for the query. Unfortunately, such grade is unknown at query-optimization time, so we need to rely on estimates to approximate it.

To determine the grade G for the filter condition f for e, we find the largest (or a close-to-largest) grade G such that the selectivity of f is at least $\frac{k}{O}$, where O is the number of objects in the repository. If the selectivity estimates used to determine G are accurate (see Section 3.2) and the independence assumption holds for e, then f is likely to retrieve the desired top-k objects, based on cost and cardinality estimates derived as in relational-model optimization as described above. However, in a realistic setting the selectivity estimates might not be completely accurate, which might result in f retrieving fewer or more than k objects. In case the number of objects retrieved is less than k, we say that query f needs to be *restarted* using a lower value for G, and the process repeats until we retrieve at least k objects.

We now present the algorithm *Rank*, which takes as input the number of objects desired k, a ranking expression e, the desired number of objects k, and the number of objects in the database O, and produces the top-k objects for e using selectivity statistics. *Rank* relies on two auxiliary functions, *FilterGrade*, which finds a suitable grade G for the filter condition used to compute the query results [4], and *FilterMap*, which simply maps a ranking expression to a filter condition that is equivalent "in structure," for a given

[4] In [9] we presented a different strategy for identifying grade G; our experimental evaluation of this alternative strategy revealed that it is comparable to or less efficient than the version that we present here. Furthermore, the older strategy suffers in performance when the distribution of grades varies significantly across attributes. Therefore we do not discuss the strategy from [9] any further in this paper due to space limitations.

grade. These two auxiliary functions are defined below.

Algorithm *Rank*(ranking expression e; objects desired k; objects in database O)

// Returns top-k objects for e among the O objects in database.

1. $reqK = k$ //Number of objects requested; might be adjusted later if restarts needed.

2. G=*FilterGrade*$(0, 1, e, reqK, O)$ // Identify search grade.

3. f=*FilterMap*(e, G) // Build filter condition equivalent to e "in structure" using grade G.

4. Use algorithm in Section 3 to find set of objects M that satisfy filter condition f

5. If $|M| < k$: // Not enough objects retrieved; need to *restart* query.

6. If $|M| > 0$: // Some objects retrieved.

7. $reqK = \lceil reqK \cdot \frac{reqK}{|M|} \rceil$ // Increase number of objects requested, to get lower search grade.

8. $newG =$*FilterGrade*$(0, 1, e, reqK, O)$

9. Else: $newG = G^2$ // No objects retrieved; object grades "squeezed" in $[0, G)$ range.

10. $G = \min\{newG, G - \epsilon\}$ // Decrease G by at least a small constant $\epsilon > 0$, for termination.

11. Go to step 3.

12. Else: Return k objects from M with highest grade for e // Enough objects retrieved; done.

Steps 5–11 handle the case where the original filter condition f with associated grade G did not manage to identify k or more objects. In this case, the query needs to be restarted, as explained above. This undesirable scenario is due to inaccurate selectivity estimations. We distinguish two cases: (1) If f matched k' objects with $0 < k' < k$ (steps 6–8), then a new, lower grade G is computed by inflating the number of objects requested proportionally to the $\frac{k}{k'}$ ratio. (2) If f matched no objects (step 9), then all objects in the database have grades in the $[0, G)$ range. The original grade G was computed assuming that grades were distributed in the $[0, 1]$ range, so we shrink grade G to the $[0, G)$ range by multiplying it by G, the new upper bound on the object grades.

The auxiliary functions *FilterMap* and *FilterGrade* are defined next. Given a grade G, *FilterMap* maps a ranking expression e into a filter condition f with e's same basic structure such that f matches exactly those objects that have a grade of G or higher for e. *FilterGrade* implements binary search to find a grade that yields the desired selectivity for a filter condition. (Reference [13] followed a similar approach to evaluate top-k queries over relational databases.)

Function *FilterMap*(ranking expression e; search grade G)

// Maps e into a filter condition f with the same structure, such that

// any objects that satisfy f have a grade no lower than G for e.

1. If $e = \textit{Grade}(A_i, v_i)$: $f = \textit{Grade}(A_i, v_i)(o) \geq G$ // e is an atomic expression.
2. Else: // e is not an atomic expression.
3. If $e = \textit{Min}(e_1, \ldots, e_n)$: $f = (\textit{FilterMap}(e_1, G) \wedge \ldots \wedge \textit{FilterMap}(e_n, G))$
4. Else: $f = (\textit{FilterMap}(e_1, G) \vee \ldots \vee \textit{FilterMap}(e_n, G))$ // $e = \textit{Max}(e_1, \ldots, e_n)$
5. Return f

Function *FilterGrade*(grade-range bounds ℓ, h; ranking expression e; objects desired k; objects in database O)

// Binary-searches for high grade G in $[\ell, h]$ range with $\textit{Sel}(\textit{FilterMap}(e, G)) \geq \frac{k}{O}$.

1. If selectivity-estimate granularity too coarse to distinguish between ℓ and h:
2. Return ℓ //Return ℓ rather than h to help avoid restarts.
3. Else:
4. $G = \frac{\ell+h}{2}$
5. If $\textit{Sel}(\textit{FilterMap}(e, G)) < \frac{k}{O}$: $h = G$
6. Else: $\ell = G$
7. Return *FilterGrade*(ℓ, h, e, k, O)

The *Rank* algorithm maps an arbitrary ranking expression into a filter condition. Note that when a query contains a filter condition F and a ranking expression R, the query asks for k top objects by the ranking expression R that satisfy F. Using *Rank*, we can translate the problem of optimizing such a query into the problem of optimizing the filter condition $F \wedge F'$, where F' is the filter condition associated with R and $k' = \frac{k}{\textit{Sel}(F)}$. We can then apply the query-processing methodology of Section 3 over this composite filter condition. In practice, it is likely that some attribute might appear both in F and in R in the original query, as in Example 2.1. In such a case, the filter condition $F \wedge F'$ will not be independent, and hence the guarantees of Section 3.3 will not hold. However, the experimental evaluation that we report next shows that the filter-condition processing techniques of Section 3.3 perform well even when the independence assumption does not hold and the data set exhibits attribute correlation.

5 Experimental Results

In this section we report an experimental evaluation of the techniques presented in Section 3 and 4, over synthetic data. In the "default" setting of our experiments, the number of objects O in each generated data set is 10,000, and objects have 6 attributes A_i, $1 \leq i \leq 6$. We vary these and other parameters throughout our experiments.

Individual attribute scores for each object are generated in three different ways:

- *Uniform* data set: We assume that attributes are independent of each other; scores are uniformly distributed within each attribute (default setting).

- *Correlated* data set: We assume that attributes are divided in two groups so that the scores of objects for attributes within the same group are correlated; scores are uniformly distributed within each attribute. We use this data set to study the performance of our algorithms when independence assumptions do not hold.

- *Gaussian* data set: We assume that attributes are independent of each other; scores are generated via five overlapping multidimensional Gaussian bells [39].

We build exact selectivity estimates over the generated data with information at a grade granularity of 0.01. We also report experiments over selectivity estimates that do not represent the data accurately.

For each attribute A_i, the probe cost $PC(a_i, p)$ to check condition a_i associated with A_i for p objects is defined to be equal to the number of objects probed p times the cost of an individual probe c_i (i.e., $PC(a_i, p) = p \cdot c_i$). We assume the search cost $SC(a_i)$ to be linear in the number of objects retrieved for condition a_i: $SC(a_i) = \mathit{Sel}(a_i) \cdot O \cdot d_i$, where O is the number of objects in the data set and d_i is the cost of retrieving one object. In our default setting, both c_i and d_i are chosen randomly from the $[1, 10]$ range.

The filter conditions that we use in our experiments have exactly one atomic condition for each of the available attributes; the grade associated with each of these atomic conditions is chosen randomly from the $[0, 1]$ range. The ranking expressions also involve all attributes, and ask for $k = 10$ objects in the default setting of the experiments.

Our default setting for the different experiment parameters is summarized in Table 2. We now report on experimental results for the default setting and when varying the different parameters. For our

Parameter	c_i	d_i	k	O	n	*Selectivity Granularity*	*Data Set*
Default Value	$[1, 10]$	$[1, 10]$	10	10,000	6	0.01	*Uniform*

Table 2: Default setting of some experiment parameters.

experiments we measure the cost of processing a query q, *Cost(q)*, as:

$$Cost(q) = \sum_{i=1}^{n} d_i \cdot Retrieved(A_i) + \sum_{i=1}^{n} c_i \cdot Probed(A_i)$$

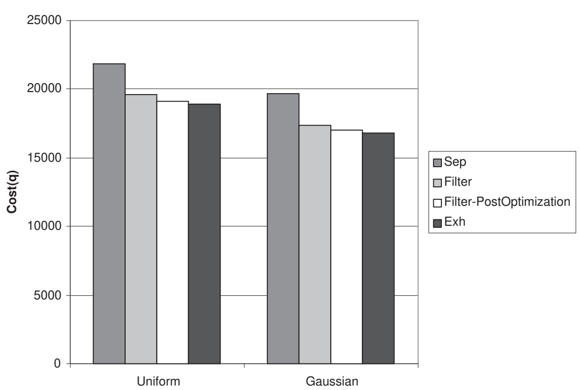
where $Retrieved(A_i)$ is the overall number of objects retrieved via *GradeSearch* over attribute A_i (including restarts and counting multiply retrieved objects) and $Probed(A_i)$ is the number of objects probed for attribute A_i (including restarts and assuming we never probe an object on the same condition twice, but rather keep this information to save probes).

5.1 Filter Conditions

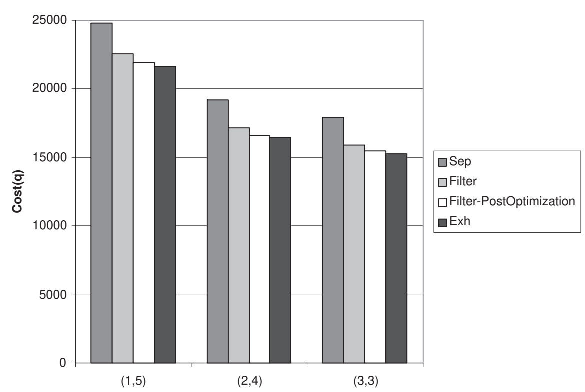
In this section we report experimental results on query processing strategies for filter conditions. Throughout this section, we use a conjunctive filter condition $a_1 \wedge \ldots \wedge a_n$, where a_i is an atomic filter condition involving attribute A_i. For our experiments, we ran 1,000 queries and averaged their results. We compare the following strategies:

- *Filter*: Strategy *Filter* is the search-minimal algorithm of Section 3.
- *Filter-PostOptimization*: Strategy *Filter-PostOptimization* is the algorithm that results from applying the post-optimization step of Section 3.4 over *Filter*.
- *Sep*: Strategy *Sep* is determined by first choosing the best atomic conditions 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. *Filter* differs from *Sep* in that the probing costs are taken into account when choosing the conditions on which to search.
- *Exh*: Strategy *Exh* exhaustively considers at query-planning time all possible non-empty subsets of the atomic conditions to retrieve the objects, and then probes the remaining conditions optimally according to the cost and selectivity statistics. This strategy does not restrict the search space to search-minimal executions as do *Filter* and *Sep*.

Figure 1 shows the performance of the four techniques for conjunctive queries for the default parameter setting, on data sets generated using different grade distributions. The correlated data sets consist of three different sets in which we divided the attributes into two groups so that an object's grades for attributes within the same group are correlated. The groups are defined as follows: (1, 5): one group has one attribute and the other five attributes; (2, 4): one group has two attributes and the other four attributes; and (3, 3): both groups have three attributes. For all data sets, *Filter* performs better than *Sep*, showing that considering probing costs when evaluating search-minimal executions results in lower query costs. *Filter-PostOptimization* gives results close to the *Exh* technique, in which all combinations of plans are considered. Interestingly, *Filter-PostOptimization* then allows to have close-to-optimal results without considering all execution plans, which can be expensive. Results for the *Gaussian* distribution are slightly better than for the *Uniform* distribution, since fewer objects tend to satisfy the selection conditions. For the correlated data sets –over which the independence assumption underlying the construction of the algorithms does not hold– all techniques have better performance when the attributes are evenly split into two groups: this configuration results in fewer probes being performed, as objects can be discarded more easily.



(a) Comparison of the different techniques for the *Uniform* and *Gaussian* data sets.



(b) Comparison of the different techniques for the *Correlated* data sets.

Figure 1: Comparison of the different techniques for data sets generated using different grade distributions.

Effect of the Number of Attributes: We studied the effect of the number of attributes in the filter condition. As the number of attributes increases, the selectivity of the conjunctive query, which then consists of more conditions, decreases. This results in turn in fewer objects being considered and in lower query-execution costs. We do not show these plots because of space limitations.

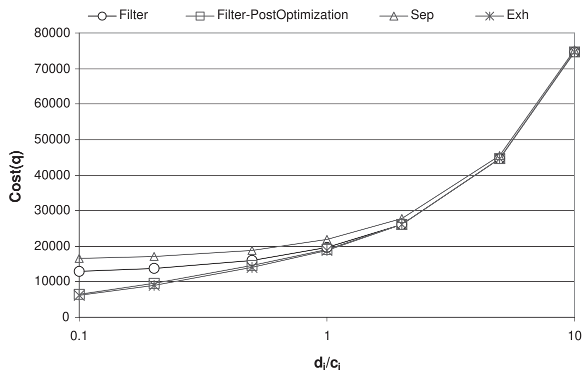


Figure 2: Effect of the cost ratio d_i/c_i on the cost of processing filter conditions.

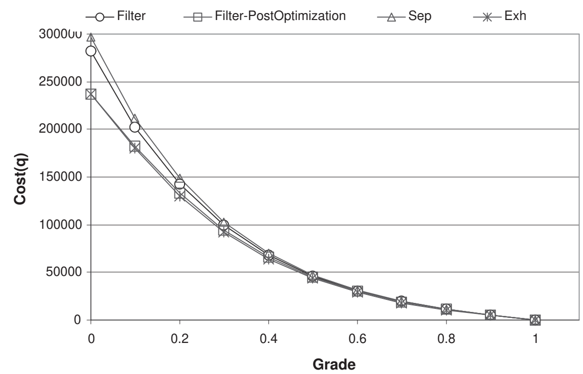


Figure 3: Effect of the condition grades on the cost of processing filter conditions.

Effect of the Cost Ratio: Our algorithms of Section 3.3 rely on cost estimations to select a query plan. Additionally, the post-optimization step of *Filter-PostOptimization* compares the relative cost of searching and probing for attributes that do not belong in the search-minimal condition set to make further optimization choices. We now study the effect on the query processing cost of the relative values of c_i, the cost of probing one object, and d_i, the cost of retrieving one object using *GradeSearch* (Figure 2). The probing cost c_i is chosen from $[0, 1]$, while the range of values of d_i varies from $[0.1, 1]$ to $[10, 100]$. As expected, when d_i increases, the overall cost of a query increases as well since retrieving objects becomes more expensive. When the d_i/c_i ratio is high, the cost of retrieving objects dominates: all techniques tend to select a plan that minimizes the number of objects retrieved using *GradeSearch*. Hence probes are favored because they are relatively inexpensive. In contrast, when the d_i/c_i ratio is low, retrieving objects via the *GradeSearch* interface is less expensive than using probes. *Exh* and *Filter-PostOptimization*, both of which consider plans with more search attributes than strictly necessary, are then cheaper than *Filter* and *Sep*, which only consider search-minimal executions.

Effect of the Condition Grades: Figure 3 studies the effect of the selectivity of a query on its cost. For these experiments, all atomic conditions in the query have the same associated grade, and we vary this grade from 0 to 1. When the grade is low, many objects satisfy the filter condition and have to be processed, resulting in high cost. In contrast, when the grade condition is high, the selectivity of the query is low and so is query processing cost.

A clear conclusion from the experiments above is that *Filter* is consistently more efficient than *Sep*: this conclusion highlights the benefits of considering the probe costs in addition to the search costs during query optimization. Another conclusion is that *Filter-PostOptimization* is significantly more efficient than *Filter*: in fact, its simple post-optimization step makes *Filter-PostOptimization* almost indistinguishable from the exhaustive-search *Exh* algorithm in our experiments. We have performed experiments over disjunctive filter conditions as well, which we do not report for space limitations: processing such queries always involves searching on *all* atomic conditions via *GradeSearch*, with no probes. Therefore, the techniques in Section 3.3 are all equivalent for disjunctive queries.

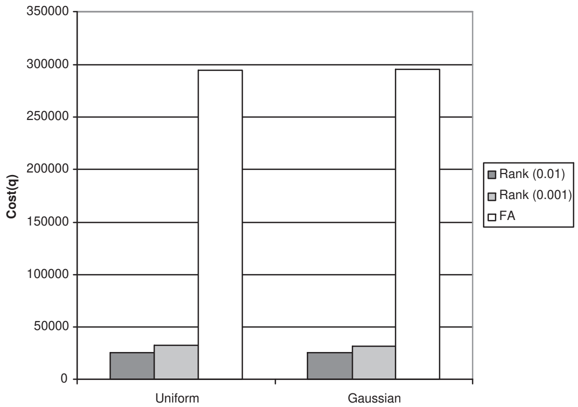
5.2 Ranking Expressions

In this section we report experiments on query-processing strategies for ranking expressions. In Section 4.3 we presented *Rank*, an algorithm to map the execution of a ranking expression into the execution of a filter condition. We now compare *Rank* experimentally with Fagin's algorithm (Section 4.2), to which we will refer as *FA*. For the filter processing part of the *Rank* algorithm (Step 4 of the algorithm, in Section 3.3), we use algorithm *Filter-PostOptimization*. Our experiments use two ranking expressions over the six attributes defined above:

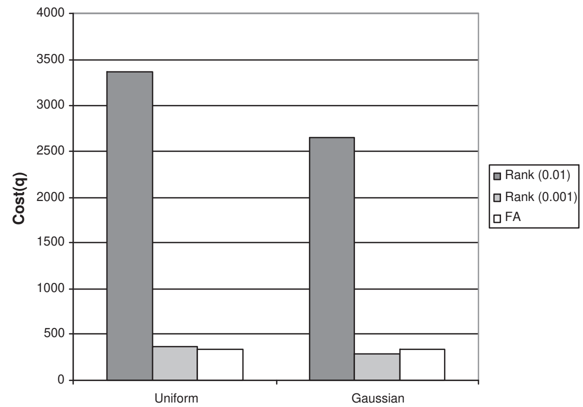
- R_{Min}: *Min*$(a_1, a_2, a_3, a_4, a_5, a_6)$
- R_{Max}: *Max*$(a_1, a_2, a_3, a_4, a_5, a_6)$

The goal of this section is to demonstrate that our heuristic technique for mapping ranking expressions to a filter condition compares favorably experimentally to Fagin's algorithm which carries optimality guarantees. Recall that the key strength of our approach is the unifying framework for answering queries involving both filter conditions and ranking expressions.

Figure 4 presents results for *Rank* and *FA* for the default settings over both *Uniform* and *Gaussian* data sets. We present results for *Rank* for two different values of the "granularity" of the selectivity estimates: 0.01 and 0.001. Figure 4(a) shows that *Rank* outperforms *FA* for the R_{Min} query. Using detailed analysis, we traced the reasons for our efficiency. First, *Rank* uses statistics on selectivity estimates (via *Filter-PostOptimization*) to decide on which conditions to search and on which to probe. This results in retrieving fewer objects than *FA* in these experiments, although the average smallest grades seen by both *Rank* and *FA* are close (the average grade G used by *Rank (0.01)*, including restarts, is 0.67235 while the average lowest grade seen by *FA* for each attribute is 0.685709 (for the *Uniform* data set). Second, *Rank*



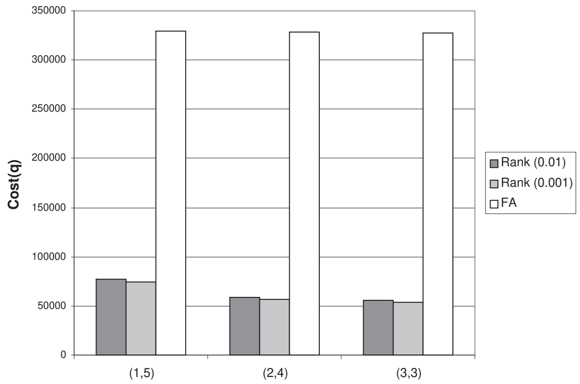
(a) Comparison of *Rank* and *FA* for R_{Min} over the *Uniform* and *Gaussian* data sets.



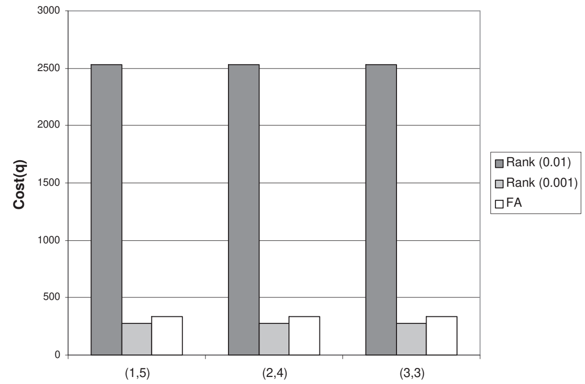
(b) Comparison of *Rank* and *FA* for R_{Max} over the *Uniform* and *Gaussian* data sets.

Figure 4: Comparison of the techniques for R_{Min} and R_{Max} over data sets generated using different grade distributions.

tends to use fewer probes than *FA*: unlike *FA*, *Rank* does not compute the complete grade of each object retrieved, but rather stops probing an object as soon as the object has failed to satisfy one condition in the filter. This early termination results in significant savings in probe costs. These two key aspects of our processing explain our performance reported in Figure 4(a). The *granularity* of the selectivity estimates slightly affects *Rank*'s query cost: a too fine granularity (see results for granularity 0.01) results in more restarts and thus higher query costs; we discuss this issue in more details below. Interestingly, *Rank*'s performance on R_{Max} functions (Figure 4(b)), which involve searching on all attributes but do not require any probing, is very sensitive to the granularity of the selectivity estimates. Specifically, if the granularity of the selectivity estimate is too coarse, the grade that *Rank* uses to map the ranking expression into a filter condition might result in a condition that matches more than k objects. Our experiments confirm that *Rank (0.01)* retrieves more objects than *FA* for R_{Max}: the average lowest grade seen by *FA* using sorted access (0.999003 for the *Uniform* data set) is slightly higher than the grade G used by *Rank* (0.99 for the *Uniform* data set). Note that the queries in the default setting ask for just top 10 objects, and that 0.99 is the highest grade that *Rank (0.01)* could pick for the default selectivity-estimate granularity of 0.01 used in these experiments. The grade G associated to *Rank (0.001)* is 0.999, and the performance of *Rank* for this finer-granularity case is almost identical to that of *FA*. Figure 5 shows the corresponding experimental results for the *Correlated* data sets.

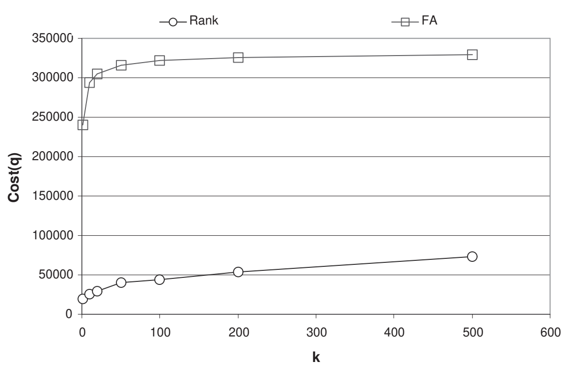


(a) Comparison of *Rank* and *FA* for R_{Min} over the *Correlated* data sets.

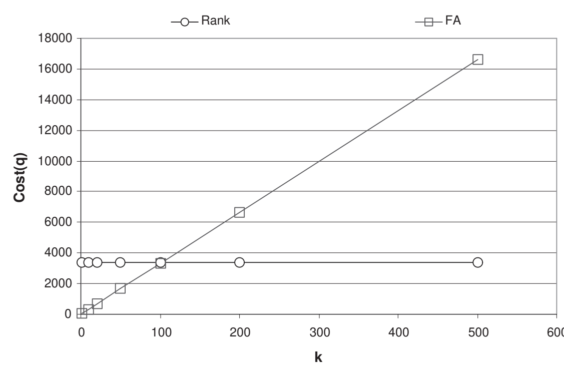


(b) Comparison of *Rank* and *FA* for R_{Max} over the *Correlated* data sets.

Figure 5: Comparison of the techniques for R_{Min} and R_{Max} over the *Correlated* data sets.



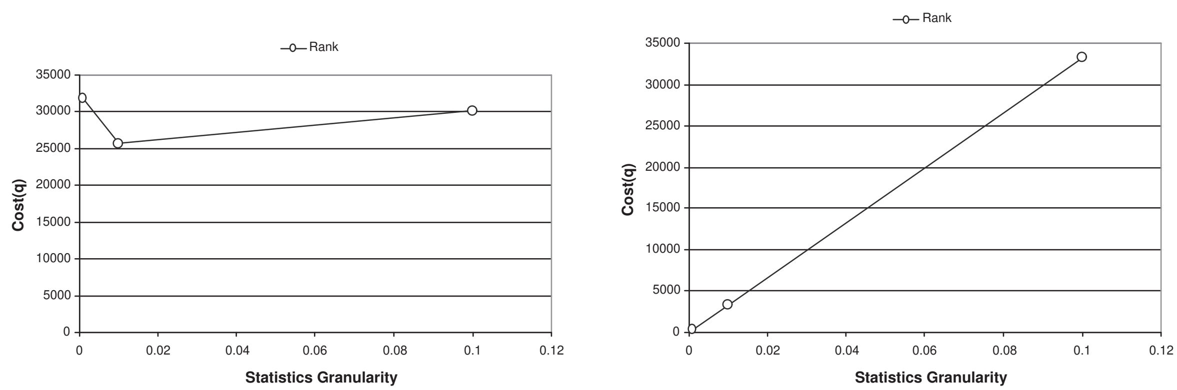
(a) Effect of the number of objects requested k for R_{Min} on the query costs of *Rank* and *FA*.



(b) Effect of the number of objects requested k for R_{Max} on the query costs of *Rank* and *FA*.

Figure 6: Effect of the number of objects requested k for R_{Min} and R_{Max} on the query costs of *Rank* and *FA*.

Effect of the Number of Objects Requested k: Figure 6 studies the effect of the number of objects requested k on the query costs of *FA* and *Rank*. Figure 6(a) shows that the cost of both techniques for R_{Min} increases slightly with k since more objects are processed to compute the query result. Figure 6(b) shows that *FA*'s performance for R_{Max} is linear in the number of objects requested k, while *Rank*'s performance is constant for the values of k that we tried: the highest grade G that *Rank* can use, given the default setting of the selectivity-estimate granularity, generally results in more objects being retrieved than needed, hence this "flat" behavior. Note that *Rank*'s cost will increase in steps each time G has to be decreased for k objects to be retrieved.

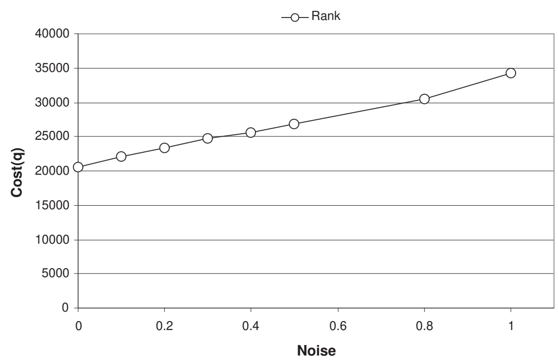


(a) Effect of the "granularity" of the selectivity estimates for R_{Min} on the query costs of *Rank*.

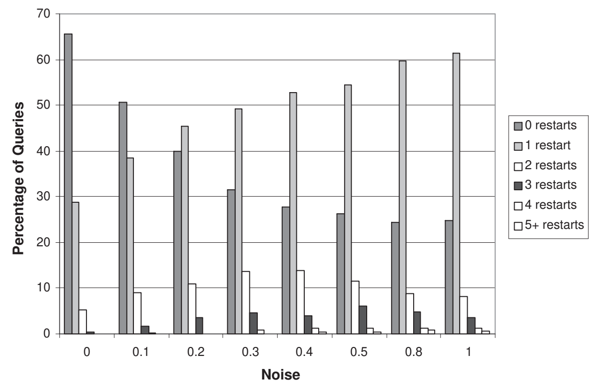
(b) Effect of the "granularity" of the selectivity estimates for R_{Max} on the query costs of *Rank*.

Figure 7: Effect of the "granularity" of the selectivity estimates for R_{Min} and R_{Max} on the query costs of *Rank*.

Effect of the "Granularity" of the Selectivity Estimates: Figure 7 studies the effect on the query costs of *Rank* of the "granularity" with which our techniques make selectivity estimates. Figure 7(a) shows that the performance of *Rank* for R_{Min} suffers if the granularity is too fine or too coarse: if the granularity is too fine, *Rank* is prone to restarts since a slight error in selectivity estimation might decrease the number of objects that satisfy the filter condition below k. (As usual, the costs reported in Figure 7 include the costs of "restarts" for *Rank*, as discussed above.) If the granularity is too coarse, *Rank* will process more objects to identify the top-k objects, since more objects are expected to satisfy the filter condition. *FA* does not use statistics on data, and is therefore unaffected by variations of the granularity of the selectivity estimates. For the setting of this experiment, *FA*'s cost is higher than 290,000. *Rank*'s performance is still much better than *FA*'s, for all granularities of the selectivity estimates that we tried. Figure 7(b) shows that the performance of *Rank* for R_{Max} improves when the granularity of the selectivity estimates becomes finer, as discussed above.

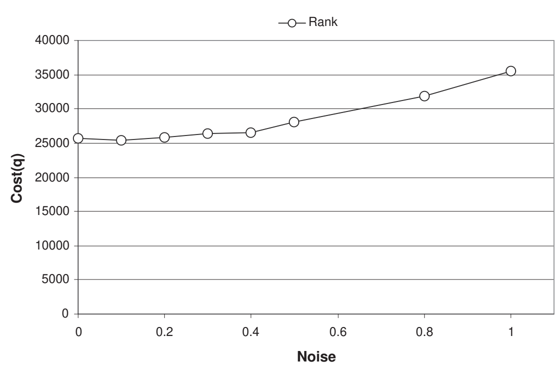


(a) Effect of data-set *noise* for R_{Min} on the query cost of *Rank* (*Gaussian* data set).

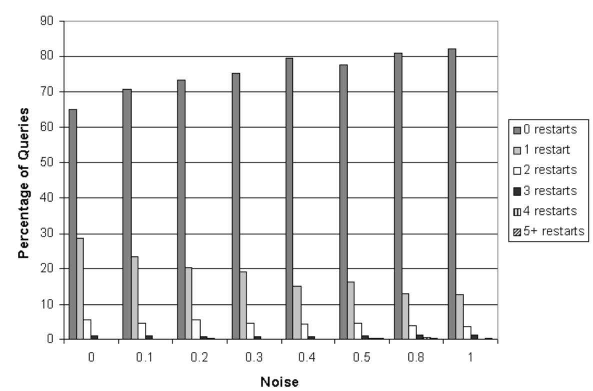


(b) Effect of data-set *noise* for R_{Min} on the number of restarts for *Rank* (*Gaussian* data set).

Figure 8: Effect on the query cost and restarts of *Rank* of the divergence of data sets and their corresponding selectivity estimates, for R_{Min} (*Gaussian* data set).



(a) Effect of data-set *noise* for R_{Min} on the query cost of *Rank* (*Uniform* data set).

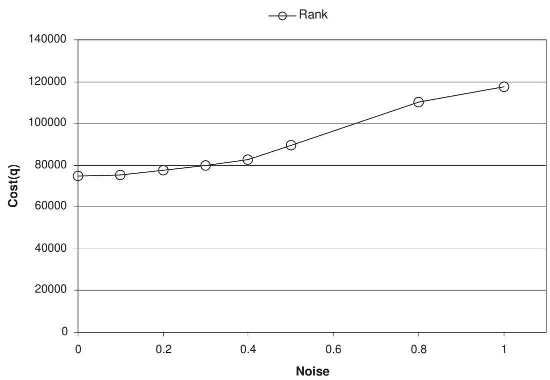


(b) Effect of data-set *noise* for R_{Min} on the number of restarts for *Rank* (*Uniform* data set).

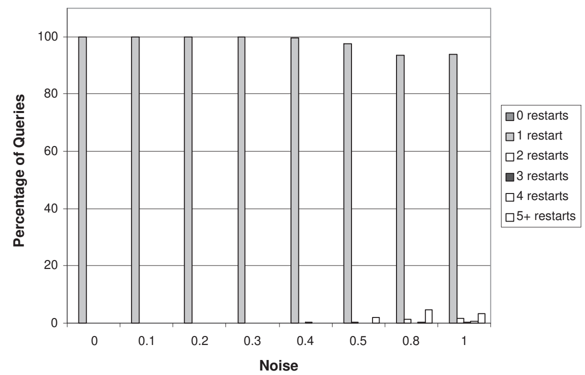
Figure 9: Effect on the query cost and restarts of *Rank* of the divergence of data sets and their corresponding selectivity estimates, for R_{Min} (*Uniform* data set).

Effect of the Selectivity-Estimate Error: *Rank* relies on selectivity estimates to map ranking expressions into filter conditions. We have already reported on the effect of the "granularity" of such estimates on the quality of the mapping. Now, we study the effect of inaccurate estimates on *Rank*. For this experiment, we use two configurations. In the first configuration, the actual data set is generated using a

Gaussian distribution with only one bell [39]. The selectivity estimates that *Rank* uses are then created using $(1 - noise) \cdot O$ objects from the actual data set and $noise \cdot O$ objects from another data set generated using a *Uniform* distribution. Thus, when the $noise$ is equal to 0, the selectivity estimates are exact, while when $noise$ is equal to 1, the selectivity estimates are highly inaccurate and based on a completely different data set generated using a different grade distribution. Results for this first configuration are shown in Figure 8. In the second configuration, the actual data set is generated using a *Uniform* distribution, and the selectivity estimates are created using $(1 - noise) \cdot O$ objects from the actual data set and $noise \cdot O$ objects from another data set generated using a *Gaussian* distribution (Figure 9). For the first configuration, selectivity estimates tend to overestimate the number of objects retrieved for a given grade. Figure 8(a) shows that the query cost is affected by the noise, and increases as expected as the $noise$ value increases (and the data set and its associated selectivity estimates become increasingly further apart). However, *Rank*'s query cost is lower than *FA*'s, even for high values of data-set *noise*. Figure 8(b) shows that the number of queries in need of restarts increases as *noise* increases, and so does the number of restarts per query. The increase in the number of restarts results from the selectivity estimates overestimating the actual number of objects retrieved for a given grade. However, the vast majority of the queries do not need to be restarted more than once, because of the grade adjustment by our "restarts" strategy, which is based on query feedback: even when $noise = 1$, only 14% of the queries require to be restarted more than once. For the second configuration, selectivity estimates tend to underestimate the number of objects retrieved for a given grade, resulting in smaller G grades. As seen in Figure 9(a), the query cost is moderately affected by the noise, since more objects than needed are being retrieved as G is lower. Figure 9(b) shows that underestimating the number of object retrieved results in fewer restarts, since more objects than estimated are actually retrieved. In summary, these results, together with those for varying selectivity-estimate granularities, suggest that *Rank* works well even with less-than-ideal selectivity estimates, especially in conjunction with our "restarts" strategy (Steps 5-11 of Algorithm *Rank*), which adjusts the search grade based on the query-result feedback from the first filter-condition execution. Figure 10 shows the corresponding experimental results for a *Correlated* (1,5) data set. For this set, there is one group of 5 correlated attributes, and the 6^{th} attribute is negatively correlated with respect to the five-attribute group. The selectivity estimates are created using $(1 - noise) \cdot O$ objects from the actual data set and $noise \cdot O$ objects from another data set generated using a *Gaussian* distribution. Since some attributes are negatively correlated, even when statistical information is correct, all queries need at least one restart. We also ran similar experiments for R_{Max} that we do not report here for lack of space, and observed that in that case *Rank* is not significantly affected by the divergence of the data set and the selectivity estimates, probably because the values of k that we tried are smaller than the expected number of objects usually retrieved using the filter condition.



(a) Effect of data-set *noise* for R_{Min} on the query cost of *Rank* ((1,5) *Correlated* data set).



(b) Effect of data-set *noise* for R_{Min} on the number of restarts for *Rank* ((1,5) *Correlated* data set).

Figure 10: Effect on the query cost and restarts of *Rank* of the divergence of data sets and their corresponding selectivity estimates, for R_{Min} ((1,5) *Correlated* data set).

6 Related Work

Our query model captures the aspects of filtering based on graded search and ranking. The concept of a graded match has been used extensively. For example, the query model in [32] allows specifying a grade of match as well as ranking. However, the processing of queries in [32] is based on searches (i.e., no probes are considered).

Many database systems have been built and prototyped with varying degrees of support for processing user-defined functions [5]. The QBIC system [29] from IBM Almaden allows users to query image repositories using a variety of attributes of the images, like color, texture, and shapes. The answer to a query is a rank of the images that best match the query values for the attributes. Another example is Cypress, a picture retrieval system built using Postgres [35] that allows a filter condition to be specified, and returns a set of objects as the answer to the filter condition. Thus, the Cypress model does not support ranking. Each object in Cypress has an image, a set of features (e.g., color histogram), an associated text and other structured information. The querying interface supports user-defined functions and 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 [24, 27, 23, 11] focuses on conjunctive selection conditions. Techniques to optimize arbitrary boolean selection conditions have been studied in [26, 25, 31]. Our work draws upon the known results in this area. (See Section 3.)

The problem of determining an optimal set of conditions to search arises naturally when optimizing single-table queries with multiple indexes [33, 28] where the problem translates into the task of identifying the appropriate set of indexes to union and to intersect[5]. By imposing the search minimality criterion, we have eliminated the need to consider index intersection and we always choose a single condition among conjuncts on which to search. This imposes implicitly the assumption that search cost is significantly higher compared to probe cost. On the other hand, we do account for non-zero probe costs, unlike [28], and are able to prove that our optimization algorithm produces an optimal search-minimal plan with low computational overhead if atomic conditions are independent. This optimization problem can also be cast as optimization of relational queries that involve joins as well as unions. As above, such a formulation fails to capture characteristics that are particular of selection queries, as exploited in our algorithms.

The information retrieval community has extensively studied the problem of ranking documents according to their expected relevance for a given query. Given a query with terms $t_1, \ldots, t_n$, a retrieval system typically retrieves the inverted lists associated with each of the terms t_i, and ranks the documents that appear in these lists [41]. If users are not interested in the entire document ranks, but only in the top document matches, some techniques avoid accessing all of the n lists associated with the terms [36].

In the context of the Garlic project at IBM Almaden [6], Fagin's work [14, 15, 16] focuses on how to evaluate queries that ask for a few top matches for a ranking expression. (See Section 4.2.) In his queries, the notions of true and false are replaced by graded matches, and boolean operators are reinterpreted to give the semantics of composition functions that take two grades of match and produce a composite grade (e.g., *Min*, *Max*). Thus, 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. Also, Fagin and Wimmers [17] discuss how to modify the scoring function to incorporate user preferences so that, say, an attribute might be twice as important to a user than the other attributes mentioned in the query. Finally, Wimmers et al. [40] describe their experience in implementing Fagin's original algorithm on Garlic. Fagin's algorithm was markedly more efficient in "joining" multiple multimedia sources compared to traditional join techniques. However, the paper also points to intrin-

[5]The problem of sequencing the order of accesses to subfiles of transposed files is also related in a similar way [2].

sic difficulties arising from heterogeneity of sources that makes establishing object identity difficult and describes the steps that were needed in Garlic to overcome these issues.

The work by Ortega et al. on the MARS system [30] developed a system for supporting ranked retrieval over image databases. One of their key contributions is an adaptation of Fagin's algorithm that has the flavor of a "merge-join" algorithm. Top-k query processing over traditional relational data has received recent attention [7, 8, 10, 13].

In this paper, we have investigated only one aspect of querying, namely that of selecting objects via a filter condition and using the ranking expression to order them. However, querying over multimedia repositories has several other dimensions that we have not addressed. For example, capturing the interrelations present in a multimedia document or in a composite multimedia object requires richer semantics and retrieval models [32, 4, 12]. Furthermore, modeling uncertainty and vagueness in data and queries is a semantic issue that is beyond the scope of this paper.

7 Summary

In this paper, we addressed the problem of *cost-based* optimization of queries over multimedia repositories. Over multimedia repositories, specifying conditions on the degree of match between values (e.g., color histograms) is an important aspect of the problem. In many of these repositories, the only way to evaluate conditions is through an index. Furthermore, we can use indexes to either evaluate a search condition or to probe a condition. We analyzed the problem of cost-based optimization of filter conditions in this framework. We have implemented a prototype retrieval system based on the ideas that we introduce in this paper. We created a sample multimedia repository consisting of objects with images and textual captions. We got the captions and images from the Digital Library project at the University of California at Berkeley, more specifically, from the Cypress project there. (See Section 6.)

We defined a space of search-minimal executions, and presented an efficient algorithm to determine the optimal choice of a search-minimal condition set for filter conditions with independent atomic conditions. Our experimental results indicate that the cost of the strategies can be significantly lowered by considering search and probe costs, compared to the cost of strategies adopted by optimizing for only the search or the probe costs separately. Although search-minimal executions minimize the number of conditions to search on, our experiments indicate that through a post-optimization step the quality of our plans is almost as good as those obtained over an exhaustive search of the plan space. Furthermore, our algorithm provides a search-minimal condition set even if the filter condition is not independent. However, optimality in such a scenario requires an exhaustive approach, as indicated by our NP-hardness result.

Another aspect of querying this type of repositories is that often the user is interested in just a few best matches for a ranking expression. A key contribution of our paper has been to show that such a ranking expression can be mapped into and executed as a filter condition with a final sorting step over just the top objects. Our thorough experimental evaluation indicates that this approach is highly efficient even when the selectivity estimates on which it relies are inaccurate, for a variety of data distributions and query scenarios.

Acknowledgments

We thank Umesh Dayal, Ron Fagin, Héctor García-Molina, Jeff Ullman, and Tak Yan for helpful discussions and comments.

References

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[2] D. S. Batory. On searching transposed files. *ACM Transactions on Database Systems*, 4(4), Dec. 1979.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R* tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM International Conference on Management of Data (SIGMOD'90)*, May 1990.

[4] E. Bertino, F. Rabitti, and S. Gibbs. Query processing in a multimedia document system. *ACM Transactions on Information Systems*, 6(1), 1988.

[5] M. J. Carey and L. M. Haas. Extensible database management systems. *ACM SIGMOD Record*, 19(4), Dec. 1990.

[6] 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*, 1995.

[7] M. J. Carey and D. Kossmann. On saying "Enough Already!" in SQL. In *Proceedings of the 1997 ACM International Conference on Management of Data (SIGMOD'97)*, May 1997.

[8] M. J. Carey and D. Kossmann. Reducing the braking distance of an SQL query engine. In *Proceedings of the Twenty-fourth International Conference on Very Large Databases (VLDB'98)*, Aug. 1998.

[9] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. In *Proceedings of the 1996 ACM International Conference on Management of Data (SIGMOD'96)*, June 1996.

[10] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *Proceedings of the Twenty-fifth International Conference on Very Large Databases (VLDB'99)*, Sept. 1999.

[11] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Transactions on Database Systems*, 24(2):177–228, June 1999.

[12] W. B. Croft and H. Turtle. Retrieval of complex objects. In *Proceedings of the 1992 International Conference on Extending Database Technology (EDBT'92)*, Mar. 1992.

[13] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top N queries. In *Proceedings of the Twenty-fifth International Conference on Very Large Databases (VLDB'99)*, Sept. 1999.

[14] R. Fagin. Combining fuzzy information from multiple systems. In *Proceedings of the Fifteenth ACM Symposium on Principles of Database Systems (PODS'96)*, June 1996.

[15] R. Fagin. Fuzzy queries in multimedia database systems. In *Proceedings of the Seventeenth ACM Symposium on Principles of Database Systems (PODS'98)*, June 1998.

[16] R. Fagin. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences*, 58(1), Feb. 1999.

[17] R. Fagin and E. L. Wimmers. Incorporating user preferences in multimedia queries. In *Proceedings of the 6th International Conference on Database Theory (ICDT '97)*, Jan. 1997.

[18] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3:231–262, 1994.

[19] C. Faloutsos and I. Kamel. Beyond uniformity and independence: analysis of R trees using the concept of fractal dimension. In *Proceedings of the Thirteenth ACM Symposium on Principles of Database Systems (PODS'94)*, May 1994.

[20] L. Gravano, H. García-Molina, and A. Tomasic. *GlOSS*: Text-source discovery over the Internet. *ACM Transactions on Database Systems*, 24(2), June 1999.

[21] W. I. Grosky. Managing multimedia information in database systems. *Communications of the ACM*, 40(12):73–80, Dec. 1997.

[22] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM International Conference on Management of Data (SIGMOD'84)*, pages 47–57, June 1984.

[23] J. M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems*, 23(2):113–157, Sept. 1998.

[24] T. Ibaraki and T. Kameda. On the optimal nesting order for computing N-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, 1984.

[25] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *Proceedings of the 1994 ACM International Conference on Management of Data (SIGMOD'94)*, May 1994.

[26] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing boolean expressions in object-bases. In *Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB'92)*, Aug. 1992.

[27] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proceedings of the Twelfth International Conference on Very Large Databases (VLDB'86)*, Aug. 1986.

[28] C. Mohan, D. J. Haderle, Y. Wang, and J. M. Cheng. Single table access using multiple indexes: Optimization, execution, and concurrency control techniques. In *Proceedings of the 1990 International Conference on Extending Database Technology (EDBT'90)*, Mar. 1990.

[29] 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, Feb. 1993.

[30] M. Ortega, Y. Rui, K. Chakrabarti, K. Porkaew, S. Mehrotra, and T. S. Huang. Supporting ranked boolean similarity queries in MARS. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 10(6):905–925, 1998.

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

[32] F. Rabitti and P. Savino. Retrieval of multimedia documents by imprecise query specification. In *Proceedings of the 1990 International Conference on Extending Database Technology (EDBT'90)*, Mar. 1990.

[33] A. Rosenthal and D. S. Reiner. An architecture for query optimization. In *Proceedings of the 1982 ACM International Conference on Management of Data (SIGMOD'82)*, June 1982.

[34] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM International Conference on Management of Data (SIGMOD'95)*, May 1995.

[35] L. A. Rowe and M. R. S. (eds.). The Postgres papers. Memorandum, UCB/ERL M86/85, University of California, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, June 1987.

[36] G. Salton. *Automatic Text Processing: The transformation, analysis, and retrieval of information by computer*. Addison-Wesley, 1989.

[37] 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 1979 ACM International Conference on Management of Data (SIGMOD'79)*, May 1979.

[38] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the Thirteenth International Conference on Very Large Databases (VLDB'87)*, Sept. 1987.

[39] S. A. Williams, H. Press, B. P. Flannery, and W. T. Vetterling. *Numerical Recipes in C: The art of scientific computing*. Cambridge University Press, 1993.

[40] E. L. Wimmers, L. M. Haas, M. T. Roth, and C. Braendli. Using Fagin's algorithm for merging ranked results in multimedia middleware. In *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems (CoopIS 1999)*, Sept. 1999.

[41] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4), 1998.