Paper Number 275

Optimizing Queries over Multimedia Repositories

Surajit Chaudhuri
Hewlett-Packard Laboratories
surajitc@microsoft.com

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

Abstract

Repositories of multimedia objects having multiple types of attributes (e.g., image, text) are becoming increasingly common. A selection on these attributes will typically produce not just a set of objects, as in the traditional relational query model (*filtering*), but also a *grade of match* associated with each object, which indicates how well the object matches the selection condition (*ranking*). Furthermore, unlike the traditional model, the users may just want a few top-ranked objects for their selection queries. In addition to the differences in the query model, another peculiarity of the 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 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 queries that ask for a few top-ranked objects can be viewed, in many cases, as that of evaluating selection conditions. Thus, both problems can be viewed together as an extended filtering problem.

1 Introduction

The problem of content management of multimedia repositories is becoming increasingly important with the development of multimedia applications. For example, digitization of photo and art collections is becoming popular, multimedia mail and groupware applications are getting widely available, and satellite images are being used for weather predictions. 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 [Fag96]. 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 the attributes of a multimedia object may only be evaluated through calls to an index. This is in contrast to a traditional database where, after accessing a tuple, all selection predicates can be evaluated on the tuple. Finally, the process of querying and browsing over a multimedia repository is likely to be interactive, and users will tend to ask for only a few best matches according to a ranking criterion.

The above observations lead us to investigate a query model with *filter conditions* as well as *ranking expressions*, and to study the cost-based optimization of such queries[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 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 good 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 a ranking expression. Previous significant work in this area is due to Fagin [Fag96], 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 without any expected loss of efficiency. We prove that our technique is expected not to retrieve more objects than the strategy in [Fag96]. Our experimental results indicate that the performance gain from processing ranking expressions as filter conditions can be substantial. This result allows us to process queries with both a filter condition and a ranking expression in a unifying framework.

[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.

We 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.)

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 [Rab90] 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 [Fag96], we address the third requirement for the query model through the notion of a *ranking expression*. The ranking expression computes a *composite grade* for an object from individual grades of match and the composition functions *Min* and *Max*. Every object has a grade between 0 and 1 for a given ranking expression. Users can 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.)

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

1. An object $o \in db$ is among the top k objects according to R_F only if o satisfies F.

2. If objects $o, o' \in db$ satisfy F, and $R(o) < R(o')$, then $R_F(o) < R_F(o')$.

The following example shows that it is not always possible to find such a ranking expression R_F for a filter condition F and a ranking expression R.

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.

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

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

∎

The above example establishes the need for both a filter condition and a ranking expression in our model.

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 [NBE+93].) 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 *Probe* atop *GradeSearch*.

Next, we briefly describe two ways to handle the type of attributes that we consider in this paper, and see how these organizations can support the interface above.

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 [Sal89]. 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 systems typically represent both documents and queries as weight vectors, where each weight corresponds to a term in the vocabulary, and produce a document rank as the answer to the queries. 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* by requesting all objects with non-zero similarity, for example.

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 [NBE+93]. One of the most popular ways of handling such attributes and queries is by using R trees [Gut84] and its variants [BKSS90, SRF87] 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 [FBF+94].

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. [RKV95] has recently presented an algorithm to find nearest neighbors on R trees. This algorithm can be used for implementing *TopSearch*.

3 Filter Conditions

In this section we will consider 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 [SAC+79].

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 two atomic filter conditions e_1 and e_2 satisfy the following: $p(e_1 \wedge e_2) = p(e_1) * p(e_2)$, where $p(e)$ is the probability that the filter condition e is true.

■

We assume that our repository requires that we use an index to evaluate every atomic filter condition. One way to process such queries is to retrieve object ids using one *GradeSearch* for each atomic condition in the filter condition, and then merge these sets of object ids through a sequence of unions and intersections. Alternatively, we can retrieve a set of object ids using *GradeSearch* for *some* conditions, 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 probing of expensive filter conditions [KMS92, HS93, IK84, KBZ86].

In this section, we first define a space of *search-minimal* executions, and sketch the cost model and the optimization criteria. Next, we describe an optimization algorithm and explain the conditions under which it is optimal. Finally, we show how we can further improve the execution plan produced by our algorithm through post-optimization, 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.

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.

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 $a_1 \vee \ldots \vee a_n$[2]. 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

[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*.

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

Example 3.3: Consider the filter condition:

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

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

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 $R(a, f) = \text{true}$. 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 the Cost Model

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

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 the objects in the repository that satisfy the condition a.

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

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

The probe cost $PC(a,p)$ depends on p, the number of probes that need to be performed. If p is large enough, it might be cheaper to implement the p probes by doing a single search on a, at cost $SC(a)$. This observation 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 [Sal89]. 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 [GGM95] 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 [FK94], 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. (See Section 5.)

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 [SAC+79]:

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

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 the 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, $|O_a| = Sel(a) * O$.

Optimizing Evaluation of Residues:

Given a residue $R(a,f)$, the task of determining an optimal evaluation for $R(a,f)$ maps to the well studied problem of optimizing the execution of selection conditions containing expensive predicates [KMS92]. (See also [IK84, KBZ86, HS93].)

If $R(a,f)$ is a conjunction of atomic conditions $a_1 \wedge \ldots \wedge a_n$, we need to sort the a_i's to minimize the cost of probing the conjunction. We may then order the atomic conditions in increasing *rank* order, where the rank of the condition a_i is $\frac{Sel(a_i)-1}{c_i}$, assuming that $PC(a_i, p) = c_i * p$ for some constant c_i. Then, given p, we can calculate the cost $PC(R(a,f),p)$ as follows. For simplicity, we assume that $a_1 \ldots a_n$ represents the increasing rank ordering of the conjuncts.

$$PC(R(a,f),p) = \sum_{i=1}^{n} S_i$$

where $S_i = Sel(a_1) * \ldots * Sel(a_{i-1}) * p * c_i$. This result is well known and was observed in the database context by [IK84, KBZ86]. 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:

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

∎

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 [KMS92]. 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'.

∎

The proof of optimality of $SM_f(f)$ depends on the fact that the given filter condition f is independent. Nonetheless, with the following simple modification, the above algorithm still provides 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 result in an optimal search-minimal condition set.

Example 3.14: Let us 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 the fact that the general optimality problem is intractable, as the following theorem shows.

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. This is why we may introduce a post-optimization step that locally replaces probes on one or more conditions by the corresponding searches. We describe one such post-optimization strategy below.

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. ∎

Next, merging the result sets from different atomic conditions also presents choices for optimization: (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 of merging results from multiple searches. (2) The merging order is also influenced by the cost of detecting and eliminating duplicate objects, and the size of the answer sets resulting from searches. Evaluating alternatives and determining their effect on the cost of execution remains a 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 recently presented [Fag96]. 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, with no expected performance loss. This result 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 the number of objects retrieved when processing a ranking expression as a filter condition can be considerably smaller than when processing the ranking expression using the algorithm in [Fag96].

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: Let us 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

Recently, Fagin presented a novel approach to processing a query consisting of a ranking expression [Fag96]. In this section we briefly describe his approach. In the next section we see how, under certain assumptions, we can do no worse than his algorithms by processing ranking expressions using a modified version of our techniques of Section 3.

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

Fagin has proved the important result that 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.

Assuming independence of the subexpressions, the expected number of objects that are retrieved from each of the expressions $a_1, \ldots, a_n$ is $L = k^{\frac{1}{n}} \cdot O^{1-\frac{1}{n}}$, where O is the number of objects in the repository. In other words, a request for top k objects for R results in an expected top L object retrievals over each of the subexpressions $a_1, \ldots, a_n$.

Now, consider a ranking expression $R = Max(a_1, \ldots, a_n)$, 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, Fagin's algorithm requests exactly k objects from each of the subexpressions a_i, $i = 1, \ldots, n$. It follows that there are k top objects for R among these $k \cdot n$ objects.

In the following section, we map the processing of a ranking expression into the processing of a filter condition.

4.3 Processing Ranking Expressions as Filter Conditions

In this section we show that we can process ranking expressions, without any expected loss of efficiency, as a modified filter condition, and by considering the space of search-minimal executions for filter conditions. Later, we show that the above result provides for an elegant and uniform treatment of ranking expressions and filter conditions.

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 flat 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 show that:

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

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

Example 4.2: Consider a ranking expression:

$$e = Min(Grade(a_1, v_1), Grade(a_2, v_2))$$

where a_i is an attribute, and v_i a constant value. We want two objects with the top grades for e. Now, suppose that we can somehow find a grade G 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:

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

By processing f using the strategies in Section 3, we obtain all of the objects with grade G or higher for a_1 and v_1, and for a_2 and v_2. Therefore, we obtain all of the objects with grade G or higher for the ranking expression e. If there are enough objects in this set (i.e., if there are at least two objects), then we know we have retrieved the top objects that we need to answer the query with ranking expression e.

Now, consider a ranking expression:

$$e' = Max(Grade(a_1, v_1), Grade(a_2, v_2))$$

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 expression 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'. If there are at least two of 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.

One way to determine the grade G for the filter condition f for e is to find any G (preferably the largest) such that the selectivity of f is at least $\frac{k}{O}$, where O is the number of objects in the repository. For any such G, the expected number of objects retrieved using the filter condition f will be at least k. However, since G is determined using selectivity estimates that could not be completely accurate, we could retrieve fewer than k objects by using f. In this case, we continue to lower grades $G' < G$ until we retrieve at least k objects.

Another way to find a grade G is to use Fagin's algorithm for processing ranking expressions. (See Section 4.2.) Given the number k of objects desired, we can determine the expected number of objects L_i that his algorithm will retrieve using the atomic expression e_i, for each i. We can then "translate" these L_i into grades g_i such that the expected number of objects with grade at least g_i for the atomic expression e_i is at least L_i. Finally, we combine all the g_i's to determine the grade G for the filter condition.

We now present the algorithm `Grade_Rank`, which takes as input the number of objects desired k, a ranking expression e, and selectivity statistics, and produces the grade G for the filter condition to process e:

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

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

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

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

Example 4.3: Consider the expression $e = Min(e_1, e_2)$ of Example 4.2, where $e_i = Grade(a_i, v_i)$. Suppose that the number of objects in the repository is $O = 100$. We want a top object for e, i.e., $k = 1$. The expected number of objects that Fagin's algorithm will retrieve from both e_1 and e_2 is $L_1 = L_2 = (k * O)^{\frac{1}{2}} = 10$. Suppose that the largest grade g_1 such that $Sel(e_1, g_1) \geq \frac{L_1}{O}$ is $g_1 = 0.2$. Similarly, $g_2 = 0.3$. Then, the algorithm above propagates these two grades to the whole expression e, yielding $G = \min 0.2, 0.3 = 0.2$. We use this value of G in the filter condition f with which we will process e (Example 4.2):

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

▮

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

Example 4.3: (cont.) Suppose that instead of f we used the following filter condition f_w (for "wrong") for processing e:

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

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

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

Now we will show that if we process a ranking expression by using a filter condition with grade G as determined by algorithm `Grade_Rank`, we can expect to retrieve no more objects than Fagin's algorithm, under some assumptions on the repositories. This result allows us to translate the ranking expressions into filter conditions, and to use the processing strategy of Section 3.

Definition 4.4: A repository is *continuously graded* if for every atomic expression e and integer L there exists a grade g such that $Sel(e, g) \times O = L$, where O is the number of objects in the repository. ▮

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

The following important theorem follows for any continuously graded repository:

Theorem 4.5: *Let R be an independent ranking expression over a continuously graded repository with O objects, and k be the number of objects desired. Let F be the filter condition associated with R that uses the grade G computed by algorithm `Grade_Rank` for R and k. Then:*

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

Proof: We will first prove that $Sel(F) \times O \geq k$. Given the ranking expression R, we first propagate the number of objects that we need from each subexpression using Fagin's algorithm. Therefore, each subexpression r of R has one such number associated with it, k_r. Also, let $FC(r, g)$ be the filter condition associated with ranking expression r and grade g. (In particular, $F = FC(R, G)$.) We will show that `Grade_Rank` assigns a grade g to each subexpression r such that $Sel(FC(r, g)) \times O \geq k_r$. In particular, $Sel(F) \times O \geq k$, because $F = FC(R, G)$.

First, if $r = a$, where a is an atomic expression, the result follows immediately because the repository is continuously graded.

Now, consider the case $r = Min(r_1, \ldots, r_n)$. Algorithm `Grade_Rank` assigns grade g_i to r_i, $i = 1, \ldots, n$, and grade $g = \min_{i=1,\ldots,n} g_i$ to r. We also know that $k_{r_i} = k_r^{\frac{1}{n}} \cdot O^{1-\frac{1}{n}}$. From the inductive hypothesis, $Sel(FC(r_i, g_i)) \times O \geq k_{r_i}$, $i = 1, \ldots, n$. Since $g \leq g_i$, $Sel(FC(r_i, g)) \geq Sel(FC(r_i, g_i))$. Then, $Sel(FC(r, g)) \times O = (\Pi_{i=1}^{n} Sel(FC(r_i, g))) \times O \geq (\frac{k_r^{\frac{1}{n}}}{O^{\frac{1}{n}}})^n \times O = k_r$.

Finally, consider the case $r = Max(r_1, \ldots, r_n)$. In this case, $k_{r_i} = k_r$ and $g = \max_{i=1,\ldots,n} g_i$. From the inductive hypothesis, $Sel(FC(r_i, g_i)) \times O \geq k_r$. Assume that $g = g_j$. Then, $Sel(FC(r, g)) \geq Sel(FC(r_j, g_j))$. Therefore, $Sel(FC(r, g)) \times O \geq Sel(FC(r_j, g_j)) \times O \geq k_r$.

We now prove that there is a search-minimal execution for F such that it is expected to retrieve no more objects than Fagin's strategy is. We use induction to prove that subexpression r of R has one such search-minimal execution for $FC(r, g)$, the filter condition associated with r and grade g, as `Grade_Rank` determines.

First, if $r = a$, where a is an atomic expression, the result follows immediately because the repository is continuously graded.

Now, consider the case $r = Min(r_1, \ldots, r_n)$. Algorithm `Grade_Rank` assigns grade g_i to r_i, $i = 1, \ldots, n$, and grade $g = \min_{i=1,\ldots,n} g_i$ to r. Assume that $g = g_j$. Then, by the inductive hypothesis, there is a search-minimal execution for r_j that is expected to retrieve no more objects than Fagin's strategy is. We can expand this execution for the entire $FC(r, g)$ by probing for the remaining $FC(r_i, g_i)$ conditions. This new execution is search-minimal, and retrieves exactly the same objects as the one for r_j.

Finally, consider the case $r = Max(r_1, \ldots, r_n)$. Algorithm `Grade_Rank` assigns grade g_i to r_i, $i = 1, \ldots, n$, and grade $g = \max_{i=1,\ldots,n} g_i$ to r. Fagin's algorithm retrieves objects for each r_i, and so does our algorithm. The result follows from the inductive hypothesis, and the fact that $g_i \leq g$, for $i = 1, \ldots, n$. ∎

Example 4.3: (cont.) Suppose that $Sel(e_1, 0.2) = Sel(e_2, 0.3) = \frac{L}{O} = 0.1$. Then, $Sel(f) \geq 0.1 \times 0.1 = 0.01$. Then, the expected number of objects satisfying f is at least $0.01 \times O = 1 = k$.

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

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

Theorem 4.5 applies to continuously graded repositories. If a repository is not continuously graded, or if the statistics that we keep on the repository are not as finely grained as to allow for modeling the repository as continuously graded, then our result does not hold. However, Section 5 shows that the approach that we outlined in this section still is a desirable one when the assumptions of Theorem 4.5 do not hold strictly.

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

5 Experimental Results

To test the results that we presented in the previous sections, we ran a set of experiments using data that we generated synthetically. We assumed a database of 1,000,000 objects, with five attributes each. Out of these five attributes, A_1 and A_2 are text attributes handled by vector-space search engines [Sal89]. The remaining three attributes, A_3 through A_5, are defined over images, and handled by R-trees indexes [Gut84]. We use these attributes to build atomic expressions $e_i = Grade(A_i, v_i)$, for fixed values v_i, $i = 1, \ldots, 5$.

To run our experiments, we define values for the selectivities and for the search and probe costs associated with each of the atomic expressions. Below we describe the assumptions that we make about these parameters. Table 2 summarizes the base parameter values that we use, which we obtained from experiments that we performed using `freeWAIS 1.0`, and from [CDY95].

- Expressions e_1 and e_2: To compute the selectivities for e_i ($i = 1, 2$) and the various grades, we define $Sel(e_i, 1.0)$, and assume that $Sel(e_i, g)$ increases exponentially with g, up to $Sel(e_i, 0.0) = 1.0$.

  To compute the search cost for e_j ($j = 1, 2$) and a grade g, we assume that the text engine that handles attribute A_j has an inverted file for A_j. Given a query, the engine retrieves the inverted lists for the query terms. Processing these lists has an initial cost c_i, a cost per entry of the inverted file that is processed c_r, and a cost c_t of transmitting the id of each matching object. Also, c_g is the cost of computing the grade for one object and the given value for the attribute.

  To compute the probing cost for e_1, we assume that there is a hash table that, given an object id o, returns the weight vector associated with o. The cost of *Probe* for an object is

Expression	Sel(e, 1.0)	D	d	c_i	c_r	c_t	c_g	c_d
e_1	3×10^{-6}	n/a	n/a	.15 s.	10^{-5} s.	.006 s.	.0002 s.	0.02 s.
e_2	7×10^{-6}	n/a	n/a	.15 s.	10^{-5} s.	.006 s.	.0002 s.	0.02 s.
e_3	n/a	9	9	.15 s.	n/a	.006 s.	.0002 s.	0.02 s.
e_4	n/a	7	3	.15 s.	n/a	.006 s.	.0002 s.	0.02 s.
e_5	n/a	10	4	.15 s.	n/a	.006 s.	.0002 s.	0.02 s.

Table 2: The base values of the parameters for the experiments, for the five atomic expressions that we consider.

then estimated as the sum of the initial cost c_i, the cost of accessing two random disk blocks $2 \times c_d$, the cost to compute the grade for o c_g, and the cost to output the result c_t.

To compute the probing cost for e_2, we assume that there is no hash table for probing attribute A_2, and so a probe is processed as a search with the lowest grade.

- Expressions e_3, e_4, and e_5: To compute the selectivities and search costs associated with these expressions, we assume that their attributes are managed using R trees. Furthermore, we use the methodology presented in [FK94] to estimate these parameters, based on the *fractal dimension* of the set of data. The fractal dimension of a set of data is a number that can be efficiently computed, and characterizes the distribution of the data. Given a range query, the fractal dimension associated with the data allows for accurate estimates of the selectivity of the query and the number of nodes of the R tree that are accessed when processing the query.

  For our experiments, we define the real dimension D (i.e., the dimension of the corresponding feature vectors) of attributes A_3, A_4, and A_5, and their fractal dimension d, and use the methodology in [FK94].

  To compute the probing costs, we assume that A_3 and A_4 scan the entire database to look for the feature vector of the object being probed, whereas A_5 has a hash table for this task, like A_1.

In the following sections, we present experimental results using the five expressions described above and their parameter values, as summarized in Table 2.

5.1 Search-minimal Executions and Post-optimization

In this section we report experimental results on optimizing strategies for filter conditions. We first present results for search-minimal executions. (See Section 3.) We then show results for strategies resulting from a simple post-optimization of search-minimal executions. Throughout this section we use a simple conjunctive filter condition $a_1 \wedge \ldots \wedge a_5$, where a_i is an atomic filter condition involving expression e_i, $i = 1, \ldots, 5$.

We first compare the cost of the best search-minimal execution against the cost of the execution of the following strategy, which we call *Sep*. Strategy *Sep* is determined by first choosing the best atomic condition on which to search, considering the search cost and the selectivity of the conditions, but not the probe costs. Then, *Sep* probes the remaining conditions in an optimal order. Our search-minimal strategy differs from *Sep* in that the probing costs are taken into account when

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

Table 3: Comparison of the optimal search-minimal executions against executions that choose the search condition without considering the probing cost.

choosing the condition on which to search. Table 3 compares the cost of a search-minimal execution versus the cost of the *Sep* strategy. We used the base values of the parameters of Table 2, except that we varied $Sel(e_2, 1.0)$. (The selectivity of e_2 for the other grades increases exponentially from that point up to $Sel(e_2, 0.0) = 1.0$, as we explained above.) Also, we chose the grade for each of the five atomic conditions from the set $\{0.7, 0.8, 0.9\}$ to obtain $3^5 = 243$ different queries. Table 3 shows that even for the simple query form that we considered, we obtained improvements in the execution time of an overwhelming majority of the queries. For example, when $Sel(e_2, 1.0) = 3 \times 10^{-6}$, the search-minimal strategy outperformed *Sep* for 94.24% of the 243 queries. The *Sep* strategy was on average 33.65% more expensive for these queries.

Given a conjunctive query, the search-minimal executions search objects on only one atomic condition. To test whether we could gain by considering more complicated strategies that search more conditions, we ran the following experiments. We first find the best strategy to process the conjunctive query by exhaustively considering all possible non-empty subsets of the atomic conditions to retrieve the objects, and then probing the remaining conditions optimally. We call the resulting strategy *Exh*. Then, we find the best search-minimal strategy as in Section 3, and we do a simple post-optimization to it. (See Section 3.4.) In particular, we consider each condition that we need to probe (i.e., each condition other than the only one used for retrieving the objects), and compare the cost of probing against the cost of searching for the condition. If the cost of searching is less than the cost of probing the condition for the estimated number of objects that will require this probing, then we search on this condition instead of probing. Such a change implements multiple probes to the condition by executing a single retrieval of all the objects that satisfy the condition, if this is cheaper. We call the resulting strategy *PostOpt*.

To compare the cost of the *PostOpt* strategies against the cost of the optimum *Exh* strategies, we ran experiments similar to the ones of Table 3. Table 4 shows some results for the same conjunctive queries, for different values of $Sel(e_2, 1.0)$, as before. The table shows that in only very few cases did the exhaustive strategy outperform the post-optimized strategy, and when it did, the difference in cost is less than 1% on average. For example, when $Sel(e_2, 1.0) = 6 \times 10^{-6}$, for only 7.41% of the queries was *PostOpt* not optimal. The last column of the table shows the average number of conditions per query that were transformed into search conditions during post-optimization. For example, when $Sel(e_2, 1.0) = 6 \times 10^{-6}$, an average of 1.41 conditions (out of a maximum of four) where changed from probes into searches during post-optimization. In summary, for a conjunctive query like the one we are using in this section, applying a simple post-optimization to the best search-minimal strategy found as in Section 3 results in almost optimal strategies most of the time.

5.2 Ranking Expressions vs. Filter Conditions

In this section we report experiments concerning the results in Section 4.1. In that section, we showed how to map the execution of a ranking expression R into the execution of a filter condition F. According to Theorem 4.5, if a repository is continuously graded, there is a search-minimal

$Sel(e_2, 1.0)$	% Non-Optimal Queries	% Difference	Times Postopt. Applied
5×10^{-6}	0	0	1.33
6×10^{-6}	7.41	0.50	1.41
7×10^{-6}	7.41	0.49	1.41

Table 4: Comparison of the post-optimized search-minimal executions against the overall optimal executions that consider all possible subsets of the conditions to retrieve objects.

execution for F that is expected to access no more objects than Fagin's algorithm for processing R directly.

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

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

The experiments in this section use two different ranking expressions over the set of five atomic expressions that we defined above:

- R_{Min}: $Min(e_1, e_2, e_3, e_4, e_5)$
- R_{Max}: $Max(e_1, e_2, e_3, e_4, e_5)$

Figure 1 shows what grade G we should use to process the filter condition associated with R_{Min} and R_{Max} to obtain at least k objects in the answer, for different values of k. We assume that the grade granularity for computing estimates is 0.1. The fact that we do not use selectivity statistics for finer grades explains that the curves are not smooth, but have "steps." Note from the figure that R_{Max} requires significantly higher grades than R_{Min}. Therefore, in general we will have to retrieve fewer objects for R_{Max} from the conditions that we use for searching.

Figure 2 shows, for each k and for R_{Min}, the expected number of objects retrieved when we process the ranking expression as a filter condition, as a fraction of the number of objects that we expect to retrieve when processing the ranking expression using Fagin's methodology. The figure shows two curves, according to the granularity of the statistics that we have about the repository, as we discussed above. We see from the figure that even when we have only coarse statistics (grade granularity=0.1), the filter-condition strategy is expected to access less than one third as many objects as Fagin's algorithm. This fraction gets even lower when we have finer statistics (grade granularity=0.01). One key point for explaining this behavior is that the filter-condition strategy searches objects from only one of the atomic subconditions, whereas Fagin's algorithm retrieves objects from all of the five subconditions in the query that we used.

Figure 3 shows similar results for the R_{Max} ranking expression. In this case, the results are not as good as for R_{Min}, because both strategies retrieve objects from all of the five atomic subconditions. Actually, when the grade granularity is 0.1, the filter-condition strategy accesses slightly more objects than Fagin's algorithm. This phenomenon disappears when the grade granularity is 0.01, except for $k = 1$ and $k = 2$. For larger values of k, the filter-condition strategy accesses significantly fewer objects. The reason of this behavior for $k = 1$ is that there are an expected seven objects that match expression e_2 with grade one, for example. Fagin's algorithm only needs to retrieve one object per expression to get the top object for a Max expression. However, for

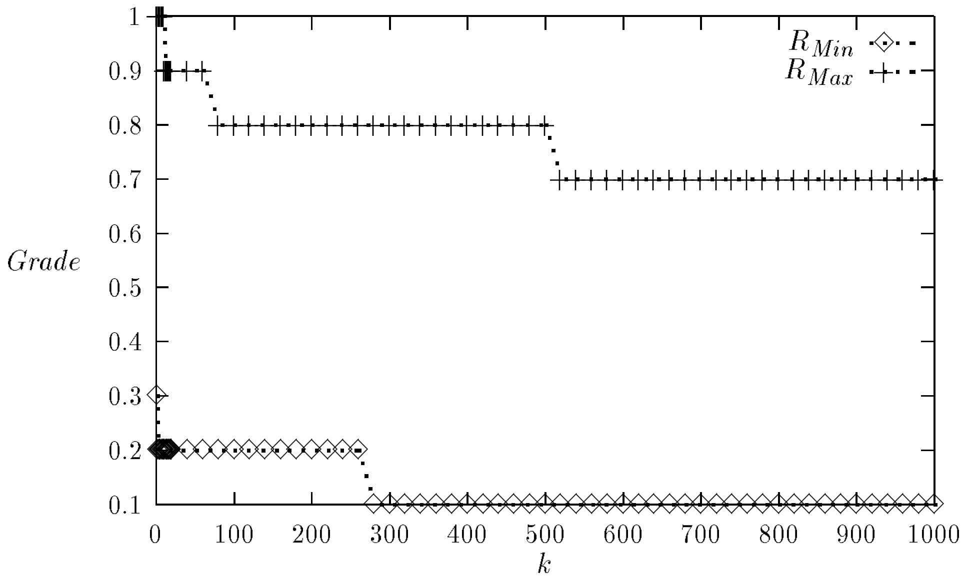

Figure 1: The grade for the filter condition associated with each of the ranking expressions, as a function of the number of objects desired k.

$k \geq 3$ the situation changes significantly. For example, for $k = 10$, the filter-condition strategy accesses only an expected 30% of the objects that Fagin's algorithm is expected to retrieve.

6 Related Work

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 [Rab90] allows specifying a grade of match as well as ranking. However, the processing of queries in [Rab90] 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 [CH90]. The QBIC system [NBE$^+$93] 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[3], a picture retrieval system built using Postgres [AS87] 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 [IK84, KBZ86, HS93] focuses on conjunctive selection conditions. Techniques to optimize arbitrary boolean selection conditions have been studied in [KMS92, KMP94, OM90]. Our work draws upon the known results in this area. (See Section 3.) However, all of the above work focuses on what we have referred to as *probing* costs, and does not consider

[3]Cypress is accessible at `http://www.elib.berkeley.edu/cypress.html`.

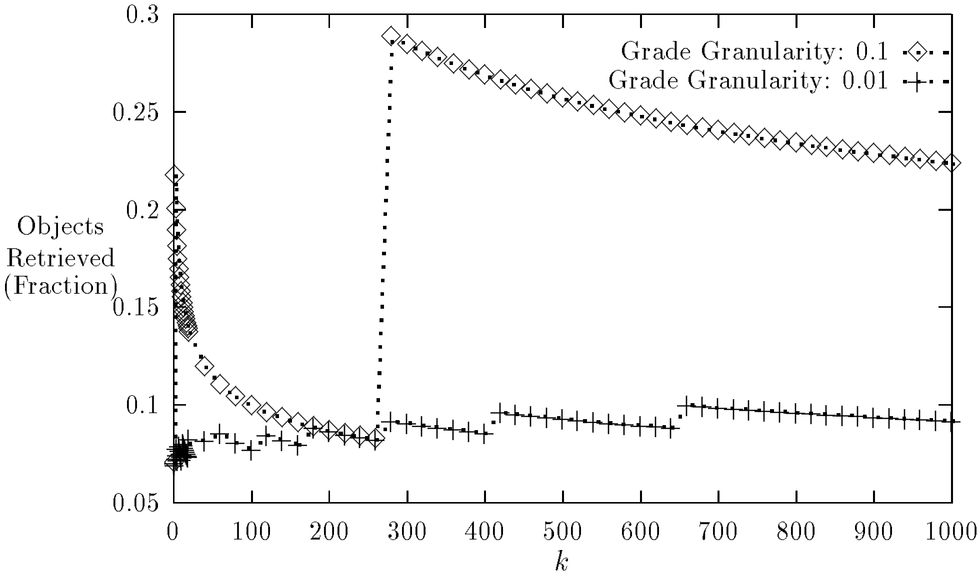

Figure 2: The expected number of objects retrieved when processing R_{Min} as a filter condition, as a fraction of the expected number for Fagin's algorithm, and as a function of the number of objects desired k.

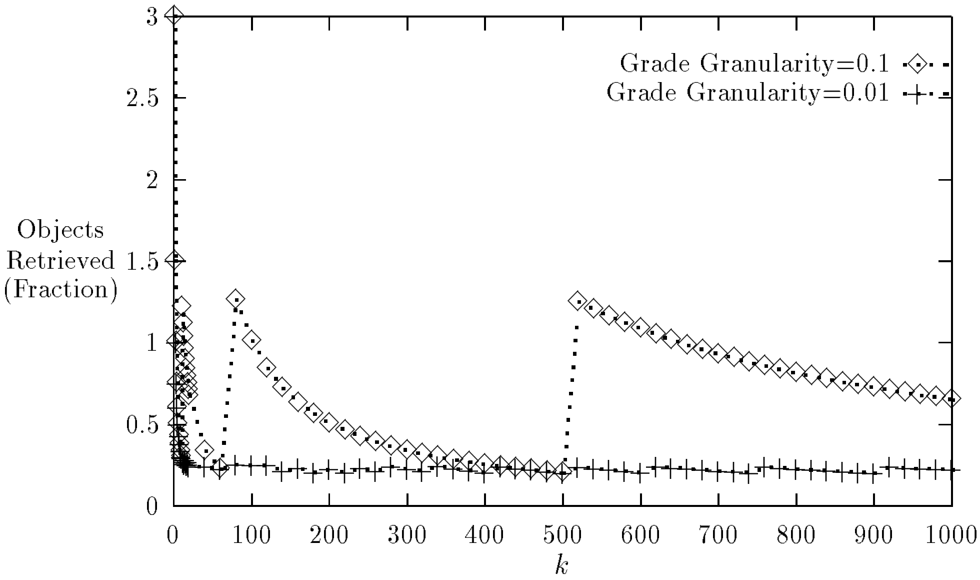

Figure 3: The expected number of objects retrieved when processing R_{Max} as a filter condition, as a fraction of the expected number for Fagin's algorithm, and as a function of the number of objects desired k.

the search costs.

On the other hand, the problem of determining an optimal set of conditions to search arises naturally when optimizing single-table queries with multiple indexes [RR82, Moh90]. The problem of sequencing the order of accesses to subfiles of transposed files is also closely related [Bat79]. However, in the above contexts, the probing cost is either zero or is independent of the predicates. Our approach to defining the execution space is similar in spirit to [Moh90], but our problem is more complex since probe costs can be significant as well as varied. When the filter condition is restricted to being a conjunction, the optimization problem can be cast as a join-ordering problem [SAC+79, IK84, KBZ86]. However, such a formulation fails to capture characteristics that are particular of selection queries. In summary, past work in this area does not consider the case where the search cost as well as the probing cost need to be considered for optimization of arbitrarily complex filter conditions containing and's and or's.

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. 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 [Sal89].

In the context of the Garlic project at IBM Almaden [CHS+], Fagin's recent work [Fag96] 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. Our contribution has been to show that ranking expressions can be evaluated efficiently by using *GradeSearch*, and that the expected number of retrievals of objects in this technique is no larger than in Fagin's approach. This result clears the way for the integration of the evaluation of filter conditions and ranking expressions.

In this paper, we have investigated only one aspect of querying - that of selecting objects via a filter condition, and using the ranking expression to order them. However, querying over multimedia repositories have 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 [Rab90, BRG88, CT92]. Furthermore, modeling uncertainty and vagueness in data and queries is a semantic issue that is beyond the scope of this paper.

7 Summary and Future Work

In this paper, we introduced and analyzed 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 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. 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 viewed "almost" like a filter condition without any expected loss of efficiency. Indeed, we demonstrated that our technique results in no more object retrievals than the approach in [Fag96], which was shown to be asymptotically optimal under broad assumptions. In fact, our experimental results indicate that the gain can be substantial. Thus, the evaluation of queries with filter conditions and ranking expressions can be viewed in a unifying framework.

Our work raises several interesting questions. We mention two of these here. First, viewing a ranking expression as "almost" a filter condition, for which we define some grade G, requires us to adjust this grade G if our estimates are proved wrong at run time. We need to analyze different alternatives for adjusting grades. Second, our experimental results showed that search-minimal executions, coupled with post-optimization, perform almost always as well as the best exhaustive-search strategy. We will explore this connection further in the future.

Acknowledgments

We thank Umesh Dayal, Héctor García-Molina, Jeff Ullman, Tak Yan, and the entire database group at HPLabs for helpful discussions and comments.

References

[AS87] Rowe L. A. and M. R. (eds.) Stonebraker. 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.

[Bat79] Don S. Batory. On searching transposed files. *ACM Transactions on Database Systems*, 4(4), December 1979.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R* tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD Conference*, pages 322–331, May 1990.

[BRG88] E. Bertino, F. Rabitti, and S. Gibbs. Query processing in a multimedia document system. *ACM Transactions on Office Information Systems*, 6(1), 1988.

[CDY95] Surajit Chaudhuri, Umeshar Dayal, and Tak W. Yan. Join queries with external text sources: execution and optimization techniques. In *Proceedings of the 1995 ACM SIGMOD Conference*, May 1995.

[CH90] M. Carey and L. Haas. Extensible database management systems. *ACM SIGMOD Record*, December 1990.

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

[CT92] W. B. Croft and H. Turtle. Retrieval of complex objects. In *Proceedings of the 1992 EDBT*, Vienna, Austria, March 1992.

[Fag96] Ronald Fagin. Combining fuzzy information from multiple systems. In 15^{th} *ACM Symposium on Principles of Database Systems*, June 1996. Also available as IBM Almaden Research Center Technical Report RJ 9980.

[FBF+94] 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.

[FK94] Christos Faloutsos and Ibrahim Kamel. Beyond uniformity and independence: analysis of R trees using the concept of fractal dimension. In 13^{th} *ACM Symposium on Principles of Database Systems*, May 1994.

[GGM95] Luis Gravano and Héctor García-Molina. Generalizing GlOSS for vector-space databases and broker hierarchies. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 78–89, September 1995.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD Conference*, pages 47–57, June 1984.

[HS93] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the ACM SIGMOD International Symposium on Management of Data*, Washington D. C., May 1993.

[IK84] T. Ibaraki and T. Kameda. On the optimal nesting order for computing N-relational joins. *ACM Transactions on Database Systems, ; ACM CR 8506 0535*, 9(3), September 1984.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zanialo. Optimization of nonrecursive queries. In *Proceedings of International Conference on Very Large Data Bases*, pages 128–137, Kyoto, Japan, August 1986.

[KMP94] A. Kemper, G. Moerkotte, and K. Peithner. Optimizing disjunctive queries with expensive predicates. *SIGMOD record*, 23(2):336, 1994.

[KMS92] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing boolean expressions in object-bases. In *Proceedings of the 18th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Vancouver*, August 1992.

[Moh90] C. Mohan. Single table access using multiple indexes: Optimization, execution and concurrency control techniques. In *EDBT 90, Venice*, 1990. Also published in/as: IBM Almaden Res.Ctr, Res.R. No.RJ7341, Mar.1990, 15pp.

[NBE+93] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, and C. Faloutsos. The QBIC project: Querying images by content using color, texture, and shape. In *Storage and retrieval for image and video databases (SPIE)*, pages 173–187, February 1993.

[OM90] M.T. Ozsu and D. Meechan. Finding heuristics for processing selection queries in relational database systems. *Information Systems*, 15(3), 1990.

[Rab90] F. Rabitti. Retrieval of multimedia documents by imprecise query specification. In *Proceedings of the 1990 EDBT*, Venice, Italy, March 1990.

[RKV95] Nick Roussopoulos, Stephen Kelley, and Frederick Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD Conference on the Management of Data*, pages 71–79, San Jose, CA, May 1995.

[RR82] A. Rosenthal and D. Reiner. An architecture for query optimization. In *Proc. ACM SIGMOD Conf.*, page 246, Orlando, FL, June 1982.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Symposium on Management of Data*, pages 23–34, Boston, MA, June 1979.

[Sal89] Gerard Salton. *Automatic text processing: the transformation, analysis, and retrieval of information by computer.* Addison Wesley, 1989.

[SRF87] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th Conference on Very Large Databases*, pages 507–518, September 1987.