A15 (Experience Track)

Using Fagin's Algorithm for Merging Ranked Results in Multimedia Middleware

Edward L. Wimmers[*] Laura M. Haas[†] Mary Tork Roth[‡] Christoph Braendli[§]

IBM Almaden Research Center
San Jose, CA 95120

March 29, 1999

Abstract

A distributed multimedia information system allows applications to access a variety of data, of different modalities, stored in data sources with their own specialized search capabilities. In such a system, the user can request that a set of objects be ranked by a particular property, or by a combination of properties. In [Fag96], Fagin gives an algorithm for efficiently merging multiple ordered streams of ranked results, to form a new stream ordered by a combination of those ranks. In this paper, we describe the implementation of Fagin's algorithm in the Garlic middleware system [C$^+$95], including a novel, incremental version of the algorithm well-suited to an environment in which users are dynamically exploring their data. While the algorithm is a simple one, designed with the distributed environment in mind, we found that the assumptions it makes about random access limit its applicability dramatically. Having implemented Fagin's algorithm, we compared it with other execution strategies for the type of self-join queries for which it might be used. The results of these experiments show that the algorithm would perform well as part of a single multimedia server, and can even be effective in the distributed environment (for this limited set of queries). Our experience provides a better understanding of an important algorithm, and exposes an open problem for distributed multimedia information systems.

1 Introduction

Multimedia data is becoming increasingly important, as hardware and software advances make using multimedia practical in a broad range of disciplines. Data of different modalities (for example, text, image and video data) are being used and accumulated in business as well as scientific applications, on the Web as well as on the intranet. With the increasing volumes of data comes the need to search that data efficiently. Querying *by content* is an important research topic in multimedia, and research prototypes and commercial products are now available for searching image, video, and audio [N$^+$93, KKOH92, PPS94, Vir, Exc]. These systems, as well as many text search engines, and other, more specialized content-based search tools, return the user a *ranked set of results.* That is, rather than simply telling the user "here are the results that meet your requirements", these search engines associate a score with each item returned that tells the user to what degree the item matches their request. Typically, they return the results ordered by this score, from

[*] *wimmers@almaden.ibm.com*
[†] *laura@almaden.ibm.com*
[‡] *torkroth@almaden.ibm.com*
[§] *christophbr@hotmail.com*

best match to worst. Often the user is only interested in the best results, and may request (or look at) only the top few.

As applications make more use of multimedia data, they also need composite objects, e.g., objects with both a text and a visual component, and need to search over these objects, possibly combining searches over the different modalities. Consider a multimedia information system which stores news articles consisting of the text of the story, and a picture. Suppose the user wants to find articles about beaches with a picture of a sunset. The system must be able to do both a text search, to find text about beaches, and an image search (perhaps on average color) to find sunset pictures, and it must be able to combine the results. The difficulty is that it is quite possible to find articles about beaches where the picture is not really at sunset, and equally possible to find articles with nice sunset pictures that have nothing to do with a beach. In other words, the same articles may be ranked quite differently by the two searches. To get the user useful results, some function will have to be applied to produce a single score for an article from the separate scores for text and image, and return the articles ordered by this new score.

In [Fag96], Fagin proposes an efficient algorithm for merging multiple streams of ranked results, given a combining function that obeys certain properties. While there are other ways to do such a merge, his algorithm is appealing, as, with very high probability, the number of objects that must be retrieved in order to get the top few results is sublinear in the total number of objects in the database. This is in contrast to naive algorithms which, even to get only the top candidate, would have to retrieve and rank all the objects in the database, combine their scores, and sort them by the new score. The algorithm has received much favorable notice in the theoretical community, resulting in an invited keynote on the same topic [Fag98].

Fagin developed his algorithm for the Garlic multimedia middleware system [C$^+$95]. Garlic provides users with the ability to search over data stored by multiple disparate data sources, exploiting the specialized search capabilities of each source. In addition, Garlic provides users with the notion of a *complex object* that links together data from different sources. In the above example, newspaper articles might be complex objects, with the text stored in one source with a text search engine, and the pictures stored in a source providing image content search. Garlic, and systems like it, could use Fagin's algorithm to merge the results of searches over the different data sources.

However, Fagin's work makes some assumptions about the multimedia system which do not always hold in the middleware environment. Chief among these are the assumption that random access is possible, and that it is easy to know, when looking at part of an object, to what object it belongs. (In our example, given the text of the news article, it must be easy to determine from which news article this text comes). In addition, Fagin looked at a particular, rather simple class of queries, whereas a general-purpose multimedia middleware must be able to handle arbitrarily complex queries in its query language. Thus, it is not immediately obvious how Fagin's algorithm will perform, and how useful it will be, in a real middleware system environment.

To answer this question, we have implemented Fagin's algorithm in the Garlic middleware system. In this paper, we describe the implementation, including a novel, incremental version of the algorithm well-suited to an environment in which users are dynamically exploring their data. We identify a necessary condition for the application of Fagin's algorithm in a given query plan, and show that this condition effectively limits the use of Fagin's algorithm to self-joins. However, when it can be used, experiments show that it out-performs other approaches to merging and reordering ranked results. Our experience provides a better understanding of an important algorithm, and exposes an open problem for distributed multimedia information systems.

There is as yet very little work related to this topic. In addition to Fagin's original paper [Fag96] and invited follow-on [Fag98], there have been a few papers proposing extensions or modifications to the algorihm [CG96, FW97]. Other papers have looked at related issues, such as how to ensure that ranked results from different sources share a common scale [GGM97]. To the best of our knowledge, ours is the first attempt to implement this algorithm and to study its implications.

In the next section, we describe Fagin's algorithm for merging streams of ranked results. Section 3 presents an architecture for multimedia middleware systems, and describes Garlic as an example of that architecture. We give examples of the kinds of queries for which Fagin's algorithm may be applicable in Section 4. In Section 5, we discuss the implementation of Fagin's algorithm in Garlic, including our incremental version of the algorithm. Then in Section 6 we focus on the problem of random access in a middleware environment, and show how it both limits the applicability of the algorithm, and complicates its implementation. Section 7 describes some experiments with the algorithm that show that when it can be applied, the performance lives up to expectations under a broad range of conditions. Section 8 summarizes our findings.

2 Fagin's Algorithm

In this section, we briefly review via an example how Fagin's algorithm merges ordered streams of ranked results. Although Fagin's algorithm can merge n streams at a time, we use two streams for simplicity. The extension to the n-way case is straightforward.

Assume that we have two streams of data, each containing at least the identifier of an object and a score, as well as, perhaps, other attributes. We will assume in this section that the object identifiers for the two streams are from the same domain, as would be the case if they were both from a single collection at one source. A *score* is a value in the interval [0,1], where the higher the score, the more closely an object matches the criteria. Each stream is ordered by decreasing score. In addition, assume that we are given a combining function for the scores. For the algorithm to work, the function must be monotonic [Fag96][1]. Finally, assume that we are given a value, k, representing the number of results the user has requested (i.e., the user is looking for the top k results). Tables 1 and 2 show the elements of two streams of data, which we want to merge to find the top two objects (hence $k = 2$). The combining function for this example will be *average*. The algorithm proceeds in three phases: a sorted access phase, a random access phase, and a final sort.

In the sorted access phase, the algorithm reads from both input streams until it finds k matches. A match occurs when the object from one stream has already been found in the other stream, i.e., when both scores for an object have been found. In essence, this phase creates a table in memory, which has three columns:

[1] A monotonic function is one for which, if $a_i \leq b_i$ for $i = 1, 2$, then $f(a_1, a_2) \leq f(b_1, b_2)$.

OID	Stream 1 Score
a	.90
d	.85
e	.83
h	.75
j	.71
b	.66
f	.40
g	.32
c	.21
i	.17

Table 1: Example: Data in Stream 1

OID	Stream 2 Score
e	.96
f	.84
b	.83
d	.55
h	.53
j	.46
c	.38
i	.37
g	.21
a	.13

Table 2: Example: Data in Stream 2

one for the object id, one for the attributes associated with that object id in the first stream, and one for the attributes associated with that object id in the second stream. When k rows of this table are complete, this phase terminates. Table 3 shows the table that results at the end of this first phase for our example. Four elements were read from each stream before two matches were found.

OID	Stream 1 Score	Stream 2 Score
a	.90	
e	.83	.96
d	.85	.55
f		.84
b		.83
h	.75	

Table 3: Example: Table after Sorted Access

In the random access phase, the table is completed. Matches are found for those objects which were only seen in one of the two streams, by asking the source of the other stream for the score (and other attribute values) of that object. To complete the process, the algorithm applies the combining function to the pair of scores for each object, and sorts by the combined score, returning the top k objects. Table 4 shows the table for our example after random access, with the combining function computed for each row. After sorting, the top two elements, namely **e** and **b**, would be returned to the user. A proof of correctness in the general case (assuming that the combining function is monotonic) is given in [Fag96].

OID	Stream 1 Score	Stream 2 Score	Combined Score
a	.90	.13	.515
e	.83	.96	.895
d	.85	.55	.70
f	.40	.84	.62
b	.66	.83	.745
h	.75	.53	.64

Table 4: Example: Table after Random Access (with Combined Score)

Of course, there are other approaches to merging our two streams. A simple solution would be to join the two streams on the object id, using a standard join method such as nested loop. Then the combining

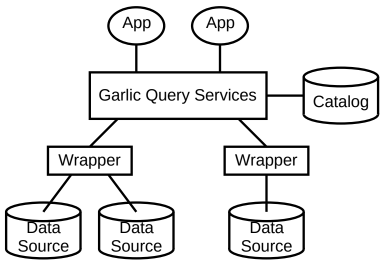

Figure 1: A Typical Middleware Architecture

function could be applied to the joined tuples, and they could be sorted by the resulting final score. However, in order to return the top k objects, this approach requires reading all the objects in the database (in order to compute the full join as input to the sort). Fagin's algorithm, by contrast, is sublinear in the number of objects in the database. In particular, for two sources[2] the cost in terms of the total number of objects in the database, N, and the number requested, k, is proportional to $\sqrt{N * k}$.

For large sets of objects, therefore, Fagin's algorithm promises much better performance than traditional join methods when the number of results requested is much smaller than the total number of objects. Thus it is appealing to add this algorithm to the "bag of tricks" of a multimedia middleware query processor.

3 Multimedia Middleware

In Section 2, we presented Fagin's algorithm without reference to the system in which it might be used. In this section, we describe the architecture of the Garlic multimedia middleware. This provides the context for Section 5, where we look at implementing Fagin's algorithm in such a system.

Figure 1 shows the Garlic architecture, which is typical of many heterogeneous database middleware systems [PGMW95, TRV96, ACPS96, SAD$^+$94]. Data sources store data and provide functions to access and manipulate their data. Multimedia middleware must potentially integrate a large variety of data sources; *e.g.*, image systems, video systems, text retrieval systems, WWW search engines, and legacy systems with specialized sets of functions. Garlic is an example of a multimedia middleware system based on an object-oriented data model. Wrappers in Garlic [RS97] describe the data in their source as objects, and Garlic refers to these objects using an OID the wrapper manufactures. This OID allows Garlic to apply methods to objects; from the OID, Garlic can determine the appropriate wrapper, and the wrapper can locate the necessary data and apply the method. Wrappers provide methods to get the value of each attribute of an object, and to encapuslate any specialized search capabilities of the source. (These methods are typically implemented as commands in the native language or programming interface of the underlying source). The wrapper also defines object *collections* (the targets of queries in Garlic). Finally, the wrapper provides a description of the source's query processing capabilities [RS97].

Different sources may vary greatly in their query processing capabilities. Most sources are able to scan a collection, retrieving OIDs and possibly other attributes as needed. Most multimedia sources can also order

[2]Again, see [Fag96], for a more complete and general analysis of the algorithm's costs.

objects in a collection according to some criteria (e.g., how blue an image is, or the distribution of colors in an image), and return the OID and score (according to that criteria) of each object. This corresponds to the "sorted access" required by Fagin's algorithm. "Random access" can be provided via method calls; we assume for the purposes of this paper that the wrapper provides a method for each type of search it supports that can return the score of an object according to that search. For example, an image wrapper providing a histogram-color search would provide a method *matches_color_histogram("filename")* that when invoked on an object returns the score for that object relative to how well it matches *"filename"*.

The Garlic middleware is essentially a sophisticated query processor. Queries asked by the user are translated into requests on the underlying data sources. Garlic's query services have two major components: a query compiler, and a distributed query execution engine. The query compiler takes a query as input and obtains an execution plan for the query through parsing, semantic checking, query rewrite, and query optimization (as in Starburst [HFLP89]). Garlic's query language is an extension of SQL along the lines of SQL3 [Mel97], that includes support for such object-oriented features as path expressions, methods and nested sets. The Garlic optimizer [HKWW97] constructs and selects an "optimal" plan for each query, based on a cost model. All alternative methods for executing a given logical operation (e.g., a join or an access to a particular collection) are considered. Once the plan has been determined by the optimizer, its execution is coordinated by Garlic's query execution engine, which passes requests to the wrappers and assembles the final query result. Garlic's execution engine is a powerful system able to perform joins, apply predicates, invoke methods, sort, aggregate, and so on. This allows Garlic to combine data from different sources, and to compensate for functionality not present in the data sources or not reflected by their wrappers.

Middleware systems allow users to exploit existing data and existing search engines, combining them to build powerful new applications. But to inter-relate data from different sources, the middleware must know about relationships between data in the different sources. For example, in the news article example of Section 1, the text is stored in one source, and the pictures in another, and yet the middleware must somehow know which text and which pictures belong together. This could happen in one of several ways. For example, the data in both the text and the image repository could share a common key, perhaps the title of the article. Or, the data in one source could refer to the other, for example, the text source might in addition to the text have the id of the picture for that text. However, if data is actually legacy data that is being combined for a new application, such links may not exist. Garlic handles this case by providing *complex objects*, new objects that can be used to record the relationship between two or more pre-existing objects. In our running example, even if the text and images had no common key, and no links to each other, a collection of news articles could be created, where each news article would be a complex object consisting of a reference to the text for that article, and a reference to the image for that article. This would allow our "text about beaches and image of sunset" query to be expressed against the collection of news articles.

4 Queries Merging Multiple Streams of Ranked Results

We have seen how Fagin's algorithm works, but for what sorts of queries can it be used? In this section, we present a sample schema and some queries against that schema, which might benefit if Fagin's algorithm were used to execute them. In our version of SQL, users can issue queries in which the ranked results of two different searches (often expressed as method calls to specialized search methods) are "joined" using a combining function (which can be an arbitrary expression), and are used in the ORDER BY clause of an SQL query to indicate that output is needed order according to the result of the combing function. [3]. If clipping the results below a certain score is desired, the WHERE clause may have a further restriction on the combined expression.

Let us look at some examples. Assume that we have two data sources, a text search engine and an image source. The text engine can rank documents via an *is_about* method, which takes a string and gives a document a score based on the number of occurences of that string (and maybe related terms) in the document. The image source provides two methods, *matches_avg_color* and *matches_histogram_color*. These methods take the name of a .gif file, and return a score indicating how well the object matches the .gif file argument. The image source exports two collections, **NewsPhotos** and **AdPhotos**. The text source exports the collections **News** and **AdCopy**. Again for the sake of the example, assume that both **NewsPhotos** and **News** objects contain the name of the article to which they belong; **AdPhotos** and **AdCopy**, however, have no common key. Finally, let **Advertisements** be a collection of complex objects having two attributes: a reference to an image, and a reference to the text of the ad. This schema is summarized in Figure 2, using a simple variant of the ODMG [Cat96] data description language.

Below we present several queries which specify multiple scored searches for which Fagin's algorithm might be useful. For simplicity, the examples use only two scored searches each, but there is no limitation in the language on the number of searches to be combined.

```
SELECT A1.OID,
       (A1.matches_avg_color(`mustard.gif´) +
       A2.matches_histogram_color(`brown+yellow.gif´))/2 as relevance
FROM AdPhotos A1, AdPhotos A2
WHERE A1.OID = A2.OID
ORDER BY relevance
STOP AFTER 10
```
(Q1)

Query Q1 finds the ten **AdPhotos** that have the most mustardy average color, and whose color histogram is closest to a certain breakdown of brown and yellow. The scoring expression appears in the select list, where it is given a name ("relevance"), for reference in the ORDER BY clause. This name "relevance" is optional; the expression could have been referred to in the ORDER BY by its position in the select list. The combining function used here is *average*, a monotonic and strict function. In Garlic's extended SQL, an ampersand before a quantifier name stands for the OID of the objects referenced by the quantifier; hence, A1.OID is the

[3]Since in SQL results can only be ordered by elements mentioned in the SELECT clause, this means that the combined scoring expression must appear in the SELECT list.

Image Wrapper Schema

```
Collections:
   AdPhotos of <Image>;
   NewsPhotos of <NewsImage>;
interface Image
   attribute String thumbnail;
   attribute String fullImage;
   method Float matches_avg_color(String);
   method Float matches_histogram_color(String);
end;
interface NewsImage
   attribute String thumbnail;
   attribute String fullImage;
   attribute String article;
   method Float matches_avg_color(String);
   method Float matches_histogram_color(String);
end;
```

Text Wrapper Schema

```
Collections:
   AdCopy of <Text>;
   News of <NewsText>;
interface Text
   attribute String body;
   attribute String author;
   method Float is_about(String);
end;
interface NewsText
   attribute String body;
   attribute String author;
   attribute String article;
   method Float is_about(String);
end;
```

Complex Object Wrapper Schema

```
Collections:
   Advertisements of <Ad>;
interface Ad
   attribute String appeared;
   attribute Date when;
   reference <Text> copy;
   reference <Image> photo;
end;
```

Figure 2: The Example Schema

OID of the **AdPhotos** ranged over by A1. The STOP AFTER clause [CK97] tells how many results should be returned. Query Q2 asks for the same results, however, it is written as a simple ordering of a single scan of the **AdPhotos** collection. Since it asks for the same result, we would expect Fagin's algorithm to be equally applicable to this query.

```
SELECT A1.OID,
       (A1.matches_avg_color(`mustard.gif´) +
       A1.matches_histogram_color(`brown+yellow.gif´))/2 as relevance
FROM AdPhotos A1
ORDER BY relevance
STOP AFTER 10
```
(Q2)

Another way to limit the number of results returned would be to clip the output at a certain level of relevance:

```
SELECT A1.OID,
       (A1.matches_avg_color(`mustard.gif´) +
       A2.matches_histogram_color(`brown+yellow.gif´))/2 as relevance
FROM AdPhotos A1, AdPhotos A2
WHERE A1.OID = A2.OID and relevance > .75
ORDER BY relevance
```
(Q3)

Of course, general predicates can also be placed on one or both streams. Query Q4, for example, finds **NewsPhotos** that match the above specification, where the article name is similar to "Haze".

```
SELECT N1.OID,
       (N1.matches_avg_color(`mustard.gif´) +
       N2.matches_histogram_color(`brown+yellow.gif´))/2 as relevance
FROM NewsPhotos N1, NewsPhotos N2
WHERE N1.OID = N2.OID and N1.article like ´%Haze%´
ORDER BY relevance
STOP AFTER 10
```
(Q4)

Our next query joins the two streams using their common key. Note that since **NewsPhotos** and **News** are from different data sources, objects will have different OIDs even if they belong to the same article. Thus, a join on OID would not be possible here. The query finds articles that are about beaches, with a picture having a color distribution similar to a sunset.

```
SELECT N.OID, P.OID,
       (N.is_about(`beaches´) +
       P.matches_histogram_color(`red-orange-yellow.gif´))/2 as relevance
FROM NewsPhotos P, News N
WHERE P.article = N.article
ORDER BY relevance
STOP AFTER 15
```
(Q5)

```
SELECT A.OID,
       binary_min(A.copy->is_about(`perfume´),
       A.photo->matches_histogram_color(`red-orange-yellow.gif´))
FROM Advertisements A
ORDER BY 2
STOP AFTER 5
```
(Q6)

Finally, Query Q6, finds the five **Advertisements** best matching the criteria "about perfume with a picture of a sunset". In this example, the combining function is a user-defined function, *binary_min*, which takes the minimum of its two arguments. This function is also monotonic and strict. The query ranges over the complex objects, **Advertisements**, using path expressions (indicated by the $\rightarrow$ in the query) to reach the associated **AdCopy** and **AdPhoto** for each. This is semantically equivalent to the explicit join in Query Q7 (assuming that every Advertisement has exactly one **AdCopy** and one **AdPhoto** associated with it). In fact, the Garlic query processor will rewrite Query Q6 as Query Q7 in order to allow more scope for optimization (more plans are possible in this formulation of the query than in the original, single collection access).

```
SELECT A.OID,
       binary_min(C.is_about(`perfume´),
       P.matches_histogram_color(`red-orange-yellow.gif´))
FROM Advertisements A, AdCopy C, AdPhotos P
WHERE A.copy = C.OID and A.photo = P.OID
ORDER BY 2
STOP AFTER 5
```
(Q7)

The above examples all involve multiple searches that return ranked results. As such, all might be candidates for evaluation using Fagin's algorithm. However, as we will see in Section 6, for some of these queries it will be difficult (if not impossible) for the query compiler to determine that the algorithm is applicable, and for still others, the algorithm cannot be used without significant extensions to either the execution engine or the algorithm itself.

5 Implementing Fagin's Algorithm

We have implemented Fagin's algorithm as part of the Garlic middleware's repertoire. In this section, we describe the basic implementation, focusing on how plans using the algorithm are generated, and on how the algorithm works at runtime. In implementing the algorithm, we realized that the need for random access severely limited the set of queries to which the algorithm could be applied in a middleware environment. The effects that this requirement has on both compilation and runtime are discussed in Section 6.

5.1 Generating Plans that Use Fagin's Algorithm

As discussed in Section 3, the Garlic query compiler starts by building up an internal representation of the query in terms of its logical operations. Each logical operation typically has a variety of algorithms associated with it, and for a given query, some or all of those algorithms may be possible implementations of that logical operation, depending on the data involved, and the query details. There are many possible ways to do a join, but typically only one way to do a particular sort. Fagin's algorithm can be viewed as an implementation of either the logical join operation, or of the ordering operation. Like a join, the algorithm merges together two (or more) streams of data. Like an ordering operation, it produces a result stream in a particular order. However, it also differs from both operations. The ordering operation reorders one incoming stream of data. Fagin's algorithm merges two or more incoming streams and reorders the result. Join algorithms are typically binary and asymmetric; which stream is the "outer" and which the "inner" affects performance and even sometimes the applicability of the algorithm. Fagin's algorithm, on the other hand, is n-ary and symmetric; there is no notion of outer or inner with it. While some join algorithms will cause the output stream to be ordered, it is a side effect of the join, not the main focus.

Ultimately, though neither was a perfect fit, we decided to treat Fagin's algorithm internally as a join, since without Fagin's algorithm in the system, these queries would be processed basically as joins, followed by a sort. We implemented Fagin's algorithm as a binary operator, as our code base thinks of joins as a binary operation, and it would have taken considerable effort to create an n-ary join. Our code base also assumes that joins are asymmetric operations, and thus considers each join method twice for a given pair of tables. Since Fagin's algorithm is symmetric, this would lead to two identical plans. Thus, we added code that builds only one plan for a symmetric algorithm.

Not all join methods can be applied in all situations. For example, Merge Join only works when the streams are ordered by the join column value, while Pointer Join[4] is only applicable when the join predicate

[4]Similar to an Indexed NestedLoop join, this join method does a direct lookup of the inner object, using in this case the

involves an OID. Fagin's algorithm, too, is only applicable under certain conditions. In particular, the algorithm relies on two key conditions: it is possible, given an object from one input stream, to find the matching attributes of the same object in the second stream (this is called the *random access condition*, because without it, the random access phase does not work)), and that the combining function used is monotonic (the *monotonicity condition*).

We will discuss when the random access condition holds in Section 6. To determine whether the monotonicity condition holds, we must determine whether the combining function used in a given query is monotonic. In Garlic, both built-in functions (such as basic arithmetic operators) and user-defined functions have a set of properties associated with them. We added a new property for monotonicity, and initialized it for our built-in functions; users can initialize it for the user-defined functions as well. Thus, when the user uses any single function as a combining function, we know instantly whether the monotonicity condition is met. However, when a complex expression using several functions (built-in or user-defined) is used as the combining function, our current implementation assumes that the resulting function is not monotonic. Ideally, the system would include some rules that would allow it to prove that certain combinations of functions were still monotonic; that would allow us to apply Fagin's algorithm to a broader range of queries.

5.2 An Iterator for Fagin's Algorithm

A Garlic execution plan is a tree of *iterators*, as in [Cat96]. Each iterator represents one algorithm of the system. For example, there are iterators for sort, nested loop join, pointer join, and so on. Iterators present a common interface; in particular, every iterator has an *advance* method, which returns the next element of its result. To get the first result of a query, the Garlic engine invokes *advance* on the top iterator of the tree. To compute the result, that iterator may have to call *advance* on its input iterators, and so on. For example, a *NestedLoopJoin* iterator would get its first element by invoking *advance* on its outer input iterator, then *advance* on its inner input iterator until it found a match.

We considered two alternative approaches for the *advance* operation for our iterator for Fagin's algorithm. In the first approach, the first time *advance* is called it runs through the entire algorithm, computing an ordered list of the top k results, and then returns the first (best) one. The second time it is invoked, it can just return the second result, with no computation required. With this approach, the entire cost of the query is paid when the user asks for the first result, and subsequent results are essentially free (up to the kth).

This approach is simple and efficient, assuming the user is going to look at all k results. In fact, it does as little work to get those k results as possible. However, it takes it a while to get the first result, and, if the user is only going to look at a subset of those results, then it has wasted effort. Since users typically don't know how many results they really want, they tend to "guess" a bound, and then browse through the list until they find what they were looking for (or something close enough). For these users, the "do-it-all-up-front" approach is not optimal.

Thus we implemented an incremental version of the algorithm. In this version, the first time *advance* is

OID for the inner tuple.

called it does the work needed for $k = 1$. The second time *advance* is called, it continues where it left off, until it has done the work needed for $k = 2$. This continues for each new call to *advance*. A side effect of this approach is that it enables Fagin's algorithm to be used cost effectively for queries in which no STOP AFTER clause is specified.

We illustrate with an example, based on that of Section 2. On the first call to *advance*, the sorted access phase produces Table 5. This table is a subset of the table produced by sorted access in the example of Section 2 (Table 3), because in that example, $k = 2$, while for this first call to *advance*, k is effectively equal to 1. We have marked with an asterisk those elements found in the sorted access phase in Stream 1, and with a dollar sign, those found in sorted access in Stream 2. This will help us on our subsequent calls to *advance*.

OID	**Stream 1 Score**	**Stream 2 Score**	
a	.90		*
e	.83	.96	*$
d	.85		*
f		.84	$

Table 5: Example: Table after Sorted Access on First Call to *advance*

The random access phase completes this table, filling in the values as shown in Table 6. Again, we show the computed combined score as well. After the table is complete, it would be sorted, and the top scoring object, **e**, would be returned.

OID	**Stream 1 Score**	**Stream 2 Score**	**Combined Score**	
a	.90	.13	.515	*
e	.83	.96	.895	*$
d	.85	.55	.70	*
f	.40	.84	.62	$

Table 6: Example: Table after Random Access on 1st Call (with Combined Score)

On the second call to *advance* to get the object with the next highest score, the algorithm continues with sorted access where it left off. In this example, it would start by getting the next element of stream 2, which in this case is **b**. It continues with sorted access, bringing in elements whether or not they were already fetched by random access, until a second match is found. Note that now a second match requires a second row in which both elements have been retrieved by sorted access (in our notation, the element would have an asterisk and a dollar sign). Sorted access on this second call to *advance* terminates with the table looking like Table 7.

The second match in this case is on **d**. While that element of stream 2 was retrieved during the random access phase on the first call to *advance*, note that it does not count as a match until it is retrieved again during sorted access on this call. At this point, random access is again used to complete the table, which ends up looking like Table 4 of Section 2. Now the element with the second highest score, **b**, is returned.

Processing continues in this manner until all k elements have been returned. Notice that this approach does do some extra work, as some elements are retrieved in the random access phase on one call to *advance*

OID	Stream 1 Score	Stream 2 Score	Combined Score	
a	.90	.13	.515	*
e	.83	.96	.895	*$
d	.85	.55	.70	*$
f	.40	.84	.62	$
b		.83		$
h	.75			*

Table 7: Example: Table after Sorted Access on 2nd Call to *advance*

that will be found again during sorted access on a subsequent call. However, this may be more than compensated for by the savings if the user chooses not to view all k results.

6 Random Access and the Applicability of Fagin's Algorithm

As noted in Section 5.1, two conditions must hold for Fagin's algorithm to be applicable: the combining function must be monotonic, and random access must be possible. In this section we focus on the challenges posed by this latter condition. In Fagin's world, the scoring expressions were viewed as predicates on a single object. Hence random access was simple: find the same OID in the other stream. But in the middleware universe, parts of objects stored in different subsystems have different identities, and queries or complex objects provide the linkage among them. For example, in Query Q5 of Section 4 above, it is the join predicate, "P.article = N.article", that tells us which element of the P stream is related to which element of the N stream. In Query Q6, the path expressions, which are later turned into join predicates during the query rewrite process, hold the key. So, given a schema, a query, and a pair of input streams, how can we determine whether, for each element of one stream there is a unique corresponding element of the other stream, and further, how can we find and retrieve that element?

6.1 Determining the Correspondence

Clearly, when we have an OID=OID predicate and each stream derives from a single collection (in other words, neither stream is the result of a join), as happens with self-joins such as Query Q1, the random access condition holds. Each incoming stream must contain the OID; thus, given any element of one stream, we can use its OID to find its mate in the other stream. However, if this were the only case in which the condition held, this would be a severe limitation on the applicability of Fagin's algorithm.

In theory, we can do somewhat better than this. To meet the random access condition for Fagin's algorithm, it is sufficient that (1) each stream have a unique attribute which, for each object in the stream, functionally determines the value of all other (selected) attributes of that object, including the score, and (2) there must be a one-to-one mapping between the unique attributes in the two streams. In other words, each stream must contain a key element, and there must be a predicate or predicates that map uniquely back and forth between the key elements of the two streams.

For example, take Query Q7 of Section 4. The query compiler will consider all possible join methods for all possible groupings of collections to find the best plan for the query. Let us look at just one possibility it would consider. Assume we already have a plan for joining **Advertisements A** with **AdCopy C**. Now we want to determine whether we can apply Fagin's algorithm to join (A $\times$ C) with **AdPhotos P**. Stream 1, the result of the earlier join of A with C, has elements of the form:

A.OID, A.copy, A.photo, C.OID, C.is_about('perfume')

while Stream 2, the result of scanning collection P, has elements of the form:

P.OID, P.matches_histogram_color('red-orange-yellow.gif')

A.OID is a key to Stream 1. Given A.OID, it is possible to determine A.copy and A.photo, as these are just attributes of the object represented by A.OID. Because of the join predicate "A.copy=C.OID", A.OID also determines C.OID, which in turn determines the score. Further, P.OID is a key to Stream 2, because the score of the photo is functionally dependent on it. Now we need the mapping between A.OID and P.OID. The join predicate "A.photo=P.OID" gets us from the OID of **Advertisements**, which is the key of Stream 1, to the OID of **AdPhotos**, which is the key of Stream 2. However, we have no predicate that, given P.OID, will return A.OID. That is, there is no way to get from the OID of **AdPhotos** back to **Advertisements**. Thus, Fagin's algorithm cannot be applied to the join of Stream 1 with Stream 2, because in the random access phase we would have no way of completing the table where elements of Stream 1 were missing.

In general, determining whether the random access condition holds is difficult, as (1) we need to find a key for each stream, (2) we need to find a mapping between the keys, and (3) we must ensure that the mapping is one-to-one. Each of these tasks can be time-consuming for complex queries; in particular, sometimes (3) is impossible without fully materializing the streams. To get a sense of the difficulties, consider that in the example above, while we quickly recognized that A.OID was the key for Stream 1, the code would have to look at each attribute of the stream, and test if it were the key by trying to establish a mapping to the other attributes. Likewise, finding the mapping between keys can be quite complex, because there does not have to be a single predicate that accomplishes the mapping. In the above example, if there were an inverse relationship (i.e, a backwards reference) between an *Image* and its *Ad*, say, an attribute of *Image* called *ad*, and the query included the predicate "P.ad = A.OID", then a mapping could be found, and Fagin's algorithm could be used to do this join.

More information would make it easier to find mappings by providing more options. To see this, note that we could, in principle, apply Fagin's algorithm to Query Q5, *if* we knew that *article* was a primary key for collections of *NewsText* and *NewsImage* objects (and that every article had both a NewsText and a NewsImage). Garlic does not currently ask for and track information on primary keys, but this would be a valuable addition to our metadata for a number of reasons, and might allow us to make greater use of Fagin's algorithm, assuming there were some way to get from the article name to the object (see Section 6.2). This is an important special case, as legacy systems often do have common keys that make it easier to interrelate data from different sources.

Finally, to be sure that the mapping is one-to-one can be the most difficult of all. So far, we have been considering unfiltered input streams, that is, streams to which no predicate has been applied. Even here, with the exception of self-joins or cases where we have information about inverse relationships, it is difficult to prove this is true. Unfortunately, predicates make the situation worse. Consider Query Q4. This simple modification of Query Q1 adds a local predicate, "N1.article like '%Haze%' ". Depending on when the predicate is evaluated, Fagin's algorithm may or may not be applicable. If the predicate is evaluated after the join, everything is fine. Likewise, if the predicate is applied to both streams before the join (which is legal, because N1.OID=N2.OID), Fagin's algorithm can still be used. The problem arises when the predicate is pushed down on one stream only (this is also legal, as elements of the other stream not meeting the predicate will be eliminated via the join predicate during the join, and in fact, such predicates are typically only applied on one stream to save evaluation cost). In this case, there will not be a one-to-one mapping between the streams, because some elements of the stream with the predicate will not pass the filter, and hence those elements when encountered in the other stream will not be matched in the filtered stream.

So we see that a key challenge for any query compiler hoping to incorporate Fagin's algorithm is determining when the random access condition holds. In Garlic today, while we have tried to write general code for finding keys and mappings, Fagin's algorithm is only found to be applicable for a fairly restrictive set of queries, typically involving self-joins. One way to make more queries amenable to using Fagin's algorithm is to introduce self-joins into them. For example, Fagin's algorithm does not apply to Query Q2, because there is no explicit join in this query. However, we noted earlier that this query returns the same results as Query Q1, which can be executed using Fagin's algorithm. As another example, consider Query Q7, which we showed above cannot use Fagin's algorithm. That query can be re-written as Query Q8. In this formulation, Fagin's algorithm *can* be used, to join the stream (A1 x C) with the stream resulting from (A2 x P). A1.OID and A2.OID are the keys of their respective streams, and the join predicate "A1.OID = A2.OID" provides the mapping.

```
SELECT A1.OID,
       binary_min(C.is_about('perfume'),
       P.matches_histogram_color('red-orange-yellow.gif'))
FROM Advertisements A1, Advertisements A2, AdCopy C, AdPhotos P
WHERE A1.copy = C.OID and A2.photo = P.OID and A1.OID = A2.OID
ORDER BY 2
STOP AFTER 5
```
(Q8)

Clearly, though, we do not want to rewrite every query as a self-join. In general, adding an extra quantifier to a query adds costs, both at compilation and execution. It is not clear under what conditions it will be beneficial to rewrite a query and use Fagin's algorithm instead of just performing the smaller join with some other algorithm. This is the sort of decision that should only be made under the control of a cost-based optimizer. Unfortunately, in Garlic rewriting is a heuristic process which occurs before cost-based optimization, and it would be difficult for Garlic to take advantage of this "trick".

6.2 Doing a Random Access

As we saw above, the need for random access makes it difficult to assess when Fagin's algorithm is applicable. It creates further difficulties at execution time, when we need to actually do the random access. We discuss these difficulties, and how Garlic does random access, in this section.

Garlic provides two means of talking to a data source (via its wrapper) during execution: method calls and queries. As mentioned in Section 3, wrappers provide methods for getting all attribute values. They also must provide methods for computing the score for any object for each kind of approximate search they support. Thus, given an object id, one way to do the random access needed by Fagin's algorithm is to invoke methods for each of the attributes needed.

This is not as easy as it sounds. Consider Query Q8 in Section 6, and suppose we want to use Fagin's algorithm to join the two streams (**Advertisements A1** × **AdCopy C**) and (**Advertisements A2** × **AdPhotos P**). The first stream, with the result of joining A1 with C, has elements of the form:

A1.OID, A1.copy, C.OID, C.is_about('perfume')

while Stream 2, the result of joining A2 with P, has elements of the form:

A2.OID, A2.photo, P.OID, P.matches_histogram_color('red-orange-yellow.gif')

Suppose during the random access phase we have to fill in a row of the table where we have the element from Stream 2, but not from Stream 1. To do this random access, we actually need to do two separate method calls, each on a different OID. Assume that the element in stream 2 has OID g. We would first invoke the method to get the *copy* attribute for **Advertisements** on object g (we know to use that OID because of the join predicate "A1.OID = A2.OID"). The result of that method call is an OID for an *AdText* object (call it **c**. Then we would invoke the *is_about* method on **c**. We know we can do that because the earlier join predicate "A1.copy = C.OID" told us how to get the third value of this element, and once we have C.OID, we can apply the method. This is a relatively simple example; if more fields of these objects are selected, keeping track of how to get the value of each attribute of the result element quickly becomes complicated. Garlic uses the method approach for random access; we build a table during query compilation that tells us what method invocations and what copying to do to fully materialize each element.

In addition to the complexity of figuring out what method calls to do to get the missing elements, the method approach has some inherent limitations due to the need to have an OID with which to start. Because of this, this approach will not work except when the predicate joining the two streams is an OID = OID predicate (in the case of a pair of predicates, as occurs with inverse relationships, one side of each of the join predicates must be an OID). This means that this approach could not extend to the situation where the predicate is an equality predicate on a primary key (see Section 6.1 above). This is because we get from one stream to another by means of a non-OID value, and given the value, even though it is unique, we cannot get the OID without asking a query. For Query Q5, for example, if we had the article name (say "TeaTime") for a **NewsPhoto**, we would have to issue the query "SELECT N.OID FROM News N WHERE N.article = 'TeaTime' " to get the OID of the matching text.

Another, more general approach to the random access phase, therefore, is to use a query to gather the needed values for a missing element. Again, wrappers provide a query interface that allows a query to be issued against their data source. In our example above, the query:

```
SELECT A1.OID, A1.copy, A1.copy, A1.copy->is_about('perfume')
FROM Advertisements A1
WHERE A1.OID = g
```
(Q9)

could be used to get all the values of the missing element. A separate query (but all of the same form) would be needed for each missing element. Of course, computing the query to be used is not necessarily any easier than computing what method calls are needed. However, this approach is certainly more general. The primary key idea could be accommodated in this approach just by changing the "WHERE" clause.

However, there are several challenges to this approach which kept us from using it in Garlic. First, it is difficult to invoke a totally new query from within the execution of another query. This new query would have to be compiled, generating a new tree of iterators, which we'd have to run, and then copy the result into the buffers for the existing tree, and continue from there. The more complex the stream, the more complex the query will be, making this a fairly inefficient path! Note that the query is a query to Garlic, not to an individual wrapper. We can't typically come up with a query that can be issued directly to the wrapper; different wrappers have different query capabilities, some of them extremely limited, so Garlic needs to be involved to make sure the query is executed correctly. For example, most wrappers cannot handle path expressions, as would be required for the query above. Also, note that the query above actually requires data from two different data sources, thus, we need Garlic involved to do the join.

Since all the queries for missing objects from a particular stream look alike, we could save some time by compiling a parameterized query (like the query above, but with a '?' where the g is). At execution time, the necessary OID value could be passed to Garlic when that query is invoked. We would still have the problem of recursively running the Garlic execution engine, but not the overhead of compilation. However, this leads to another difficulty: to get good performance on queries, Garlic needs to push as much work as possible to the data source [HKWW97]. For many data sources, simple predicates (for example, those on OIDs or primary keys) can be executed. But if the query is parameterized, the wrapper must be willing to handle the *binding* in of the parameter value [RS97]. Since this is an optional interface, many wrappers may not support it. However, if they don't, the only choice may be to retrieve a lot of extra data from the data source, and evaluate the predicate in Garlic. This could also be extremely inefficient.

Due to these assorted difficulties, we chose to use the method approach for random access in Garlic. For the simple queries to which we can apply Fagin's algorithm, methods work quite well. In general, there does not seem to be a really good solution to the random access problem (at least for complex queries) in the middleware environment. Since the sources are autonomous, the middleware cannot force them to provide a "fast path" for random access. Since every source is different, the middleware cannot count on more than a few basic interfaces (otherwise, the task of writing a wrapper becomes too complex). One concern we had in implementing Fagin's algorithm was whether the inefficiency of random access would have a negative effect on performance, as well as applicability. In the next section, we describe some of our experiments with the algorithm, and the results of this and other investigations.

7 Some Experiments

7.1 Experimental Environment

For our experiments, we ran queries against a Garlic system with three data sources, including QBIC [N$^+$93], an image server that ranks images according to user-specified color, texture and shape features. The other sources were a relational database, and an object-oriented database that stores Garlic complex objects. Given the limited applicability of Fagin's algorithm, we focused on queries that applied multiple QBIC predicates to the same collection of images, in other words, self-joins such as Query Q1. However, QBIC provides several different ways of ranking images, with very different performance characteristics. Hence, we believe that our results would extend easily to an environment with multiple collections in different sources, were the problems with applicability solved.

For example, QBIC provides two functions for ranking images by color: average color and color histogram. Both functions analyze an input image, compare it with images stored in the image database, and return those images ranked according to how well they match the input image. For average color queries, the initial computation depends on the x- and y- dimensions of the image predicate. For color histogram queries, the initial computation depends on the number of colors specified in the image predicate. A simple algorithm that is linear in the number of colors is used when only a few colors are specified, and a more complex algorithm is employed that is quadratic in the number of colors if a large number of colors are specified. By varying the number of colors in color histogram queries, it was possible to ask queries where the costs of the various scans and random accesses varied quite dramatically.

One characteristic of our experimental environment that is unrealistic is the size of the database. All of our experiments were run on a collection consisting of 547 images. Fagin's algorithm was designed for situations in which the size of the data was several orders of magnitude larger than this. In general, a larger database size will give Fagin's algorithm a much bigger advantage. However, we feel that the results described below give a good sense of the algorithm's potential, and we expect the effects described would only be exaggerated in a larger database.

In this section we will mean by *random access cost* the cost to do one random access to retrieve data from a source. By *sorted access cost* we will mean the cost to retrieve the next element in a sorted stream. We will be comparing the performance of Fagin's algorithm (FA) to pointer join followed by a sort (PJ), and occasionally nested loop join followed by a sort (NLJ). Pointer join, as mentioned earlier, behaves similarly to an indexed nested loop join. Each object in the outer collection must provide the OID of an object from the inner. Then for each object in the outer, the matching inner object is retrieved using that OID. Garlic's nested loop join applies any local predicates to the inner, then stores the result temporarily in Garlic, where it does the NLJ in the usual way, scanning the entire inner for each object from the outer.

7.2 Verifying the Predicted Behavior of Fagin's Algorithm

We began our study by looking at the performance of Fagin's algorithm for a range of queries, to see if it would exhibit the predicted $\sqrt{N * k}$ behavior. Figures 3 and 4 are typical of the results. Both figures show the time to find the top k results for k ranging from 1 to the size of the image collection (547 images). For comparison, we have also plotted the performance of a pointer join followed by a sort (the best competitive algorithm in Garlic for these queries). Note that the pointer join is an almost flat line, because the sort forces all work to be done up front, before any results are returned. In Figure 3 the two searches were both on average color. One search ranked images by how close to black they were, the other by how close to white; hence there was no correlation between the scores in the two streams. Figure 4 shows a different query, again involving two average color searches, but here the two colors were much closer to each other, so the scores from the two streams exhibit some correlation. We see that the overall curve has the same shape as that in Figure 3, but it has a smaller slope, inclining further out to the right, so that it crosses the competing pointer join curve at $k = 283$ instead of at $k = 190$ as in the no correlation case. (Before the crossover point, FA is the winner, taking less time to retrieve the best k objects than PJ; if the user is asking for that many objects or less, the system should use FA. After that point, PJ is the better choice). In both of these figures, we have also plotted the total amount of time spent on random and sorted accesses. We can see that the amount of time spent doing random accesses is much less for the case where there is some correlation in the scores. [WHRB98] shows this even more dramatically for an almost perfectly correlated search.

Note that, even for uncorrelated streams, FA handily beats out the competing pointer join up until $k = 190$ – a full 35% of the data! This is impressive for an algorithm that has as its goal to allow the user to inexpensively browse the top few percent of their results, especially given the high random access costs in our environment. We next look at this issue in more depth.

7.3 The Effect of Access Costs on the Performance of Fagin's Algorithm

Our next series of experiments focus on how Fagin's algorithm behaves as the performance characteristics of the middleware environment (especially the various costs of accessing data) vary. To avoid confusing effects, the query is identical in all of these experiments; it has two easy histogram searches with relatively uncorrelated arguments.

In the first of these experiments, we looked at what happens as the random access costs increase. This models the situation in which data sources have a very high overhead for random access, or the middleware is forced to use a high overhead strategy (Section 6.2). In this experiment, we recorded the performance of three execution strategies (Fagin's algorithm (FA), pointer join followed by a sort (PJ), and a version of nested loop join that caches the inner at the middleware, also followed by a sort (NLJ)) as we increased the average random access cost by adding an artificial delay which ranged from 50 to 200 ms. Figure 5 plots time to execute the query in seconds versus the random access delay for NLJ, PJ, and FA at three different values of k. Since NLJ does no random access, it is unaffected by the delay, hence shows constant performance. Clearly, as the delay gets enormous, NLJ will eventually beat all competitors. However, for delays less than

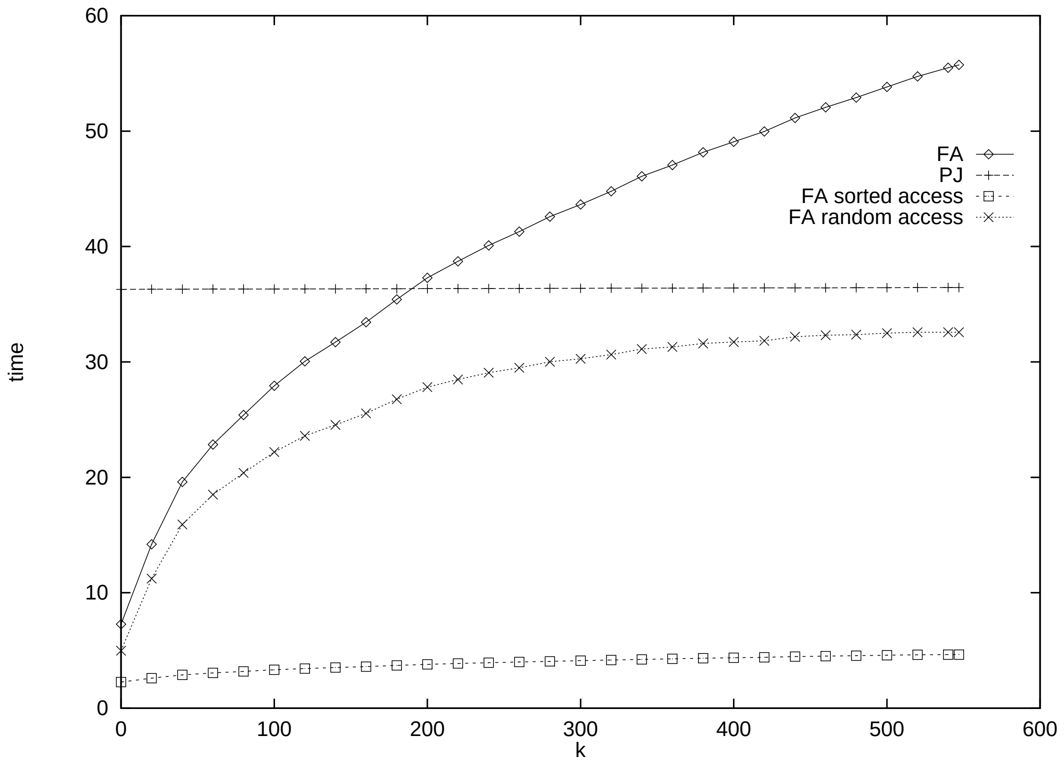

Figure 3: A Query with 2 Uncorrelated Searches

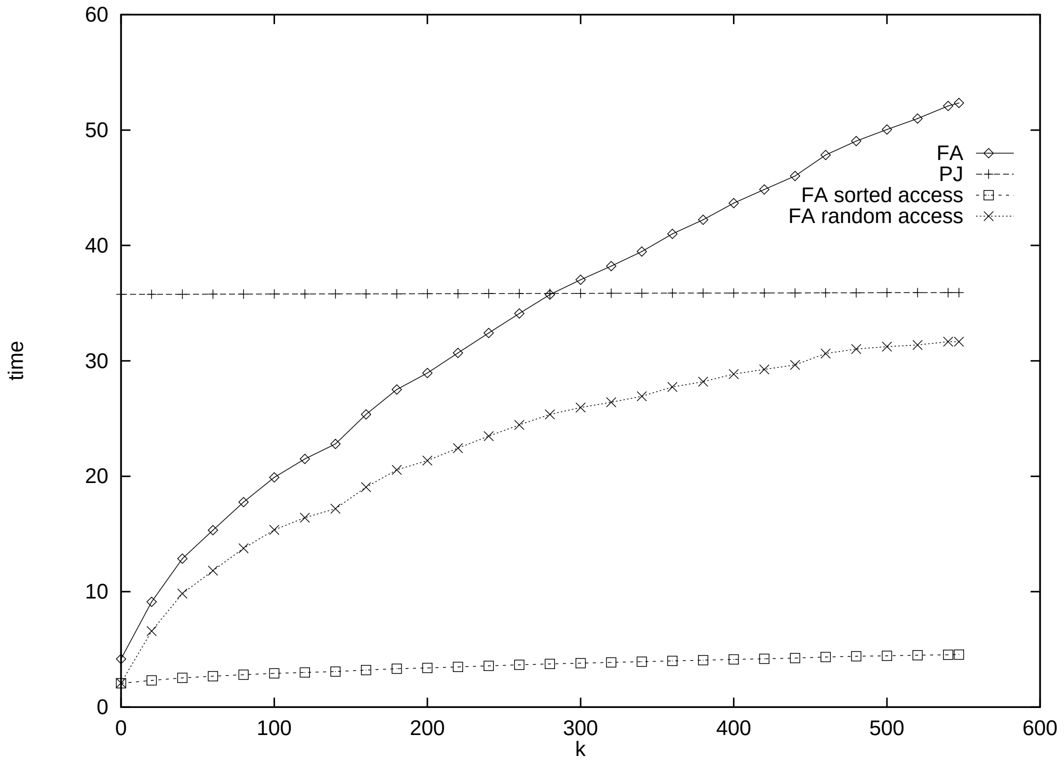

Figure 4: A Query with 2 Somewhat Correlated Searches

120 ms, PJ is a better choice, and for reasonable values of k (e.g., $k < 100$), FA will typically be faster, even for much larger delays.

In our second experiment, we varied the delay imposed on a random access for one of the two searches, to see what effect an asymmetry in the costs would have. This could happen easily if searches at one source were more expensive than at another. PJ can minimize the effects of such asymmetric delays; FA must suffer them. We compared FA to PJ for several values of k, as the delay for one search varied from 50 to 200 ms. The results are shown in Figure 6. Since NLJ does no random access, we do not include it here. For this experiment, we exploited the asymmety of PJ, letting it use sorted access (the cost of which was, somewhat artificially, kept constant) for the expensive search, and random access for the cheap search. Thus, since it does no expensive random accesses, the PJ line is essentially constant. FA, on the other hand, is slowed down dramatically by the increasing random access delay (regardless of which stream is the expensive stream).

We also examined how FA would behave in two other environments. In the first, all access costs are high. This simulates an environment in which a fast middleware processor communicates over slow lines to remote data sources. In this case, both PJ and FA slow down (of course), but FA is less affected for small values of k, and more affected at higher values. In the second environment, random access and sorted access had the same cost, to simulate how FA might perform within an individual multimedia search engine (in which random access would be much faster than in our middleware environment). Once again FA was the best choice for a wide range of k values.

7.4 Discussion

We have seen that Fagin's algorithm behaves well for a broad range of queries, and a broad range of access costs. For small k it is almost always the algorithm of choice, even in quite adverse circumstances. In our standard middleware environment, it is surprisingly effective, winning handily even for k values up to 35% of the database size, despite the slowness of random access costs. With a larger database, we would expect the savings to be even more dramatic. The algorithm is likely to be extremely useful for multimedia search engines that specialize in the sort of query for which it is intended, and if the algorithm were more broadly applicable in the middleware context, it would clearly be an important addition to a middleware engine.

Unfortunately, the algorithm as it stands has limited applicability, and this, combined with the difficulty of testing whether it is applicable, make the cost of the algorithm to query compilation more than it is worth for a general-purpose middleware. In [WHRB98], we examine several variants of the algorithm that might increase its applicability. One variant is fully general, allowing the algorithm to be used for any query; however, the applicability comes at the expense of losing all guarantees about performance. A potentially more interesting variation allows more efficiency for key-based joins, when certain hashing functions can be found. Overall, however, a widely applicable and efficient algorithm for merging ranked results in the general middleware environment remains to be found.

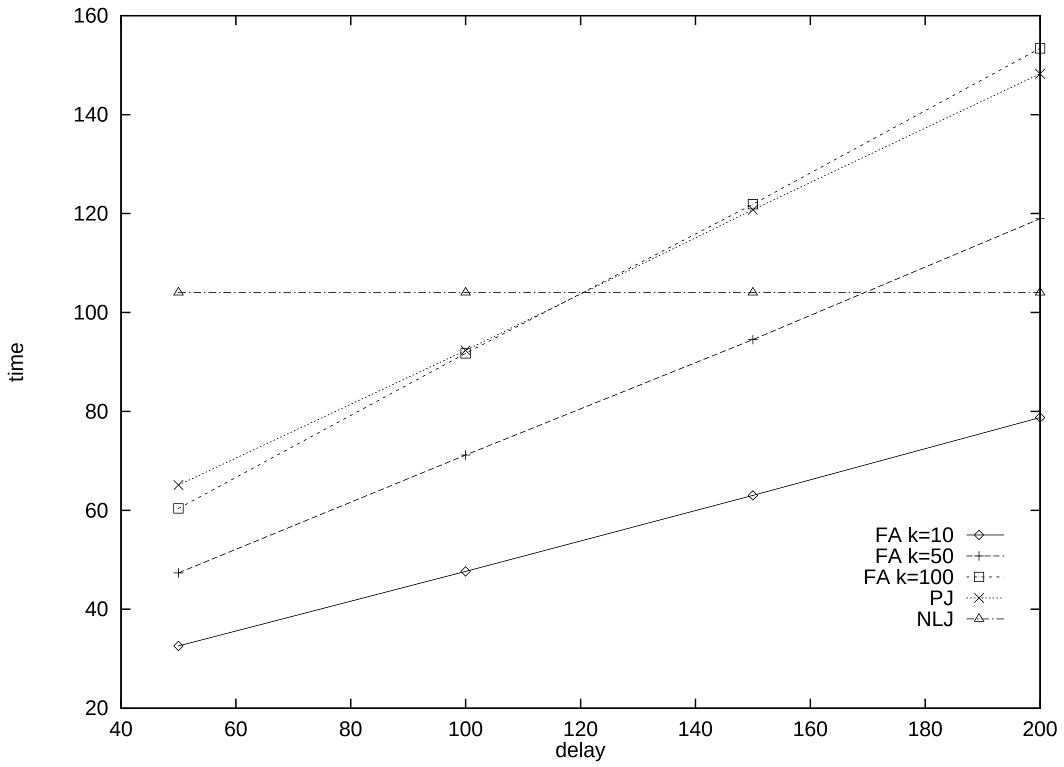

Figure 5: Increasing Delay for both Random Accesses

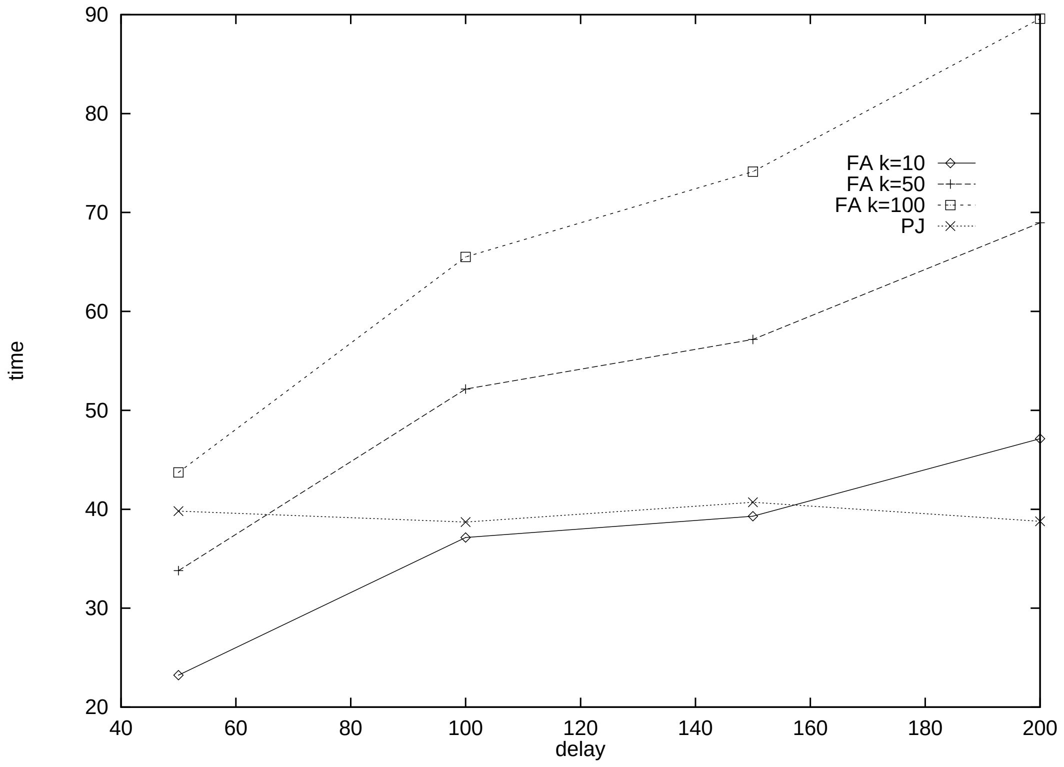

Figure 6: Increasing Delay Asymmetrically for One Random Access

8 Conclusions

We have described the implementation of Fagin's algorithm for merging ordered streams of ranked results in Garlic, a typical multimedia middleware system. We presented an incremental implementation of the algorithm which can be used even when there is no STOP AFTER clause in situations where the user is likely to stop before viewing all the data she has requested. We also showed that the algorithm can be applied to only a limited set of queries in the middleware environment, due to the lack of common object identifiers and the need for random access. However, for those queries to which the algorithm can be applied, it is often the best performing algorithm for a wide range of values of k.

Though Fagin's algorithm may not be ideally suited to the middleware environment that we consider, we believe the performance results presented in Section 7 prove that it should be seriously considered for implementation in any multimedia data source that needs to be able to apply multiple ranking predicates on a single collection of data. This would apply to many content-based search engines available today. Such systems will typically meet the conditions on which Fagin's algorithm depends.

Merging ordered streams of ranked results from different sources in multimedia middleware remains an important problem. In the future, we plan to explore some of the generalizations presented in this paper, as well as other algorithms for this problem. It is particularly important to find a good algorithm for handling the case in which the two streams share a common (non-OID) key (e.g., Query Q5), as this is a common case for the sorts of legacy systems that Garlic integrates.

9 Acknowledgements

We gratefully acknowledge the inspiration of Ron Fagin, who goaded us into this work and listened patiently to our complaints as we struggled through it. We also thank Alyssa Pritchard, whose initial implementation of this algorithm in an earlier version of Garlic exposed many of the difficulties we addressed in this implementation. Peter Haas and Peter Schwarz provided invaluable technical assistance. Markus Tresch and Hans-Joerg Schek made it possible for Christoph to come to Almaden and do his master's thesis on this topic, and provided constructive comments throughout. Finally, we thank Ron Fagin and Peter Haas for their careful reading and comments on an earlier draft of this paper.

References

[ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 137–148, Montreal, Canada, June 1996.

[C$^+$95] M. Carey et al. Towards heterogeneous multimedia information systems. In *Proc. of the Intl. Workshop on Research Issues in Data Engineering*, March 1995.

[Cat96] R. G. G. Cattell. *The Object Database Standard – ODMG-93.* Morgan-Kaufmann Publishers, San Mateo, CA, USA, 1996.

[CG96] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, June 1996.

[CK97] M. Carey and D. Kossmann. On saying 'enough already' in sql. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Tucson, AZ, USA, May 1997.

[Exc] Excalibur. www.excalib.com. web site.

[Fag96] R. Fagin. Combining fuzzy information from multiple systems. In R. Hull, editor, *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 216–226, Montreal, Canada, 1996. The Association for Computing Machinery.

[Fag98] R. Fagin. Fuzzy queries in mulitmedia database systems. In C. Faloutsos, editor, *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 1–10, Seattle, WA, USA, 1998. The Association for Computing Machinery.

[FW97] R. Fagin and E. Wimmers. Incorporating user preferences in multimedia queries. In *Proc. 6th Int'l Conference on Database Theory*, volume 1186 of *Lecture Notes in Computer Science*, pages 247–261. Springer-Verlag, January 1997.

[GGM97] L. Gravano and H. Garcia-Molina. Merging ranks from heterogeneous internet sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, 1997.

[HFLP89] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 377–388, Portland, OR, USA, May 1989.

[HKWW97] L. Haas, D. Kossmann, E. Wimmers, and J. Wang. Optimizing queries over diverse data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, 1997.

[KKOH92] T. Kato, T. Kurita, N. Otsu, and K. Hirata. Query by visual example. In *International Conference on Pattern Recognition (ICPR)*, pages 530–533, The Hague, The Netherlands, September 1992.

[Mel97] J. Melton(ed). Database language SQL - part 2: SQL/Foundation, September 1997. ISO/IEC JTC1/SC21 N11106.

[N$^+$93] W. Niblack et al. The QBIC project: Querying images by content using color, texture and shap. In *Proc. SPIE*, San Jose, CA, USA, February 1993.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. IEEE Conf. on Data Engineering*, pages 251–260, Taipeh, Taiwan, 1995.

[PPS94] A. Pentland, R. Pickard, and S. Scarloff. Photobook: Tools for content based manipulation of image databases. In *SPIE*, San Jose, CA, March 1994.

[RS97] M. Tork Roth and P. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, 1997.

[SAD$^+$94] M.-C. Shan, R. Ahmed, J. Davis, W. Du, and W. Kent. Pegasus: A heterogeneous information management system. In W. Kim, editor, *Modern Database Systems*, chapter 32. ACM Press (Addison-Wesley publishers), Reading, MA, USA, 1994.

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of DISCO. In *Proc. ICDCS*, 1996.

[Vir] Virage. www.virage.com. web site.

[WHRB98] E. Wimmers, L. Haas, M. Tork Roth, and C. Braendli. Using Fagin's algorithm for merging ranked results in multimedia middleware, August 1998. IBM RJ Number 95005.