CERIAS Tech Report 2001-82
Distributed processing of filtering queries in HyperFile
by Christopher Clifton
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Distributed Processing of Filtering Queries in HyperFile[†]

Chris Clifton[‡] and Hector Garcia-Molina

Department of Computer Science, Princeton University, Princeton, NJ 08544

Abstract

Documents, pictures, and other such non-quantitative information pose interesting new problems in the database world. We have developed a language for queries which serves as an extension of the *browsing* model of hypertext systems. The query language and data model fit naturally into a distributed environment. We discuss a simple and efficient method for processing distributed queries in this language. Results of experiments run on a distributed data server using this algorithm are presented.

1. Introduction

HyperFile is a back-end data storage and retrieval facility for *document management* applications. The goal of HyperFile is not just to store traditional documents containing text. It also supports multimedia documents containing images, graphics, or audio. In addition, it must support *hypertext* applications where documents are viewed as directed graphs and end-users can navigate these graphs and display their nodes. Another goal is to provide a *shared* repository for multiple and diverse applications. For example, it should be possible for a user running a particular document management system to view a VLSI design stored in HyperFile. Similarly, a user running a VLSI design tool should be able to refer to a document that describes the operation of a particular circuit.

Given our requirements, it makes sense to implement HyperFile as a back-end service, as shown in Figure 1. Although not essential, we do expect that in many cases applications and HyperFile will run on separate computers. This is because: (1) HyperFile represents a shared resource so it is important to off load as much work as possible, (2) the applications probably have different hardware requirements (e.g., color graphics displays) than the service (e.g., large secondary storage capacity, high performance IO bus), and (3) it enhances the autonomy of the applications.

We stress that the HyperFile "server" will often be distributed over multiple computers. In some cases, the source objects or documents will be inherently distributed over multiple nodes. For example, old papers would be placed on an archival server, whereas it makes sense to keep work in progress on the author's workstation. As a more extreme example, two geographically distant institutions may want to (transparently) share information; however neither wishes to provide space for storing the other's documents. In others cases, distribution is required to provide reliability, high performance, large capacity, and/or modularity. As we will see throughout this paper, this distribution requirement drives many of the design decisions made in HyperFile.

Given that we wish to provide a data server, the most important question is what *interface* to provide the applications. There is actually a spectrum of possibilities. At one end we have a *file interface*. In this case, the server only understands named byte sequences. The server does not understand the contents; it can only retrieve a file given its name or store a new file. From one point of view, this is a good model: it makes the data server simple, off loading all of the interpretation of the data to the application. One could even argue that it facilitates sharing because it does not impose a particular data model that may be inappropriate for some applications. On the other

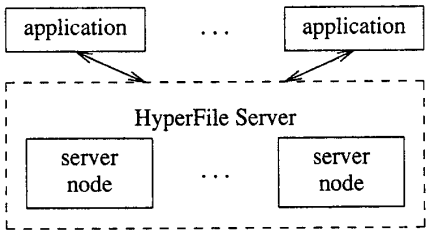

Figure 1: HyperFile *as a back-end service.*

[†] This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts Nos. N00014-85-C-0456 and N00014-85-K-0465, and by the National Science Foundation under Cooperative Agreement No. DCR-8420948. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

[‡] Work of this author supported in part by an IBM Graduate Fellowship.

hand, a file interface increases the number of server-application interactions and/or the amount of data that must be transmitted. For example, say we want to search for a book with some given properties, e.g., published between May 1901 and February 1902. Since the server does not understand publication dates, the application will be forced to retrieve many more books than are actually required. Of course, the application could also build index structures for some common queries, but then these indexes do not cover all cases, plus traversing the index structures also requires interactions with the server.

For the type of applications we are considering, we would like to have some more server search functions, while still preserving the simplicity and flexibility of a file interface. This is precisely the goal of HyperFile. The philosophy is that HyperFile will not understand the contents of objects, except for some key properties (defined by the application) that will be used for retrieval. Examples of properties may be the title of a paper, the clock speed of a particular chip, the objects that are referenced (hypertext links), or the previous version of a program (pointer to another object). Searches based on these properties will be performed by HyperFile, usually with a single request and retrieving *only* the data of interest. More complex searches (e.g., find all chips that have a race condition) will involve additional processing by the application. The fundamental idea is that HyperFile is powerful enough so that, for the applications of interest, most of the searching can be done at the server, while at the same time being straightforward enough to have a simple and efficient distributed implementation.

There are a number of distributed system issues that have driven the design of HyperFile. We outline a few here.

- Communication may be expensive. HyperFile servers may be widely separated. Therefore messages should be as small as possible, limited in number, and able to be sent using simple protocols.
- HyperFile should scale well. The system may be large, and queries may only need objects from a few nodes. Only those nodes should be involved in processing the query.
- Server nodes may be autonomous. They should not be subject to any more global control than necessary, and lack of cooperation from one node must not shut down the entire service.
- Partial results are better than none at all. If *Node A* is down, one should still be able to pose a query to *Node B*. This may not produce a complete answer to the query, but it may be adequate.

In our server interface spectrum, there are of course other options in addition to files and HyperFile. We feel that they do not meet the goals we have for a data server.[1] However, at this point we do want to stress we are not ruling out other interfaces for different applications (or even for document processing ones). As a matter of fact, other interfaces (such as an object-oriented database or a file system) could be implemented at the server next to (or even on top of) HyperFile. Our point is that HyperFile represents an interesting point in the interface spectrum, providing the right mix of facilities and simplicity for many document management applications.

The following section gives an overview of the HyperFile data model and query interface. Section 3 gives the algorithm we use for processing queries. Other issues in a distributed environment are discussed in Section 4. Finally, in Section 5 we describe a prototype implementation of HyperFile and present results from an experimental evaluation of the query processing algorithm.

2. The Query Interface

In HyperFile, objects are modeled as sets of tuples. These tuples can contain text, pictorial data, keywords, bibliographic information, references and pointers to other objects, or arbitrary bit strings. A sample set, containing (for example) a module from a Software Engineering system, is:

{ (String, "Title", "Main Program for Sort routine")
(String, "Author", "Joe Programmer")
(Text, "Description", *<Arbitrary text description.>*)
(Text, "C Code", *<Text of the Program>*)
(Text, "Object Code", *<Executable for module>*)
(Pointer, "Called Routine",
<Pointer to another object>)
(Pointer, "Library", *<Pointer to a library used by this routine>*) }

Note that tuples have three parts: A **type**, which identifies the data types of the remaining fields to HyperFile; a **key**, which is used by the application to specify the purpose of the tuple; and **data**, which can be a simple type such as a string or pointer, or complex (and not understood by HyperFile) such as a paragraph of text or the object code of a program. The possible entries in the type field are not fixed; applications can define new types. For example, an application could define **Object_Code** to be a type where the key would name the target machine. This would be a convention between applications; HyperFile would only understand **Object_Code** as a type of tuple having a string as a key, and arbitrary bits as data. The data server does not understand (or restrict) the concepts of "target machine" or "object code".

[1] We have not included a more detailed comparison with other systems here due to space constraints. A comparison of HyperFile with other systems is available in Princeton University Department of Computer Science Technical Report CS-TR-295-90, an extended version of this paper.

Tuples may contain pointers to other objects, as shown in the above example. From the viewpoint of an application, such pointers simply identify other objects regardless of location. In other words, distribution is transparent. The query processing algorithm must handle remote pointers differently from local ones; this is discussed in Section 3.

An application may use multiple HyperFile objects to store what the end user views as a single "document". For example, one text processing application may wish to store an entire paper in a single object, while another one may store each paragraph in a separate object, linking them together into sections and chapters with additional objects. This is entirely up to the application.

As stated in the introduction, our goal is to retain (as much as possible) the simplicity and flexibility of a file system. This is why our objects have such an elementary model. There is no rigid, predefined schema, and there are no object classes. Our model is similar to that of a file system with *self-describing data records*[14]. In such a system, records of a file contain tags stating what information is contained in the record.

HyperFile queries are based on the browsing techniques of *hypertext*[5]. The problem with browsing is that it is labor-intensive; selection is done by manually navigating through the data. We expand this with a query interface based on *document sets* and *filtering*. Items returned by a query are determined by the scope which would be browsed and specifications as to the contents of the desired objects. These queries consist of three parts:

- A starting set of objects in the graph-structured document repository (corresponding to the "current document" in a browsing interface.)
- A set of filtering criteria (keywords, size, etc.)
- A description of where to look: What types of links to follow (and how far) to find prospective objects.

HyperFile provides *sets of objects*. These sets are used as the starting point for queries. A set of objects is created using a basic object, with tuples containing pointers to the objects in the set. The set of objects {*A, B, C*} is simply an object containing three tuples, one of which points to each of *A*, *B*, and *C*. Figure 2 shows a set S containing three objects: *M* (from the previous example), *N*, and the library *L*. Note that *M* (which is also the example at the beginning of this Section) can be used as a set containing the library *L* and the called routine *C*.

Queries select objects which contain tuples matching certain patterns in the key field, and in some cases in the data field. In addition, queries can follow pointers in order to select new objects. There are a variety of query types: Set operations (union, intersection, etc.), basic selection operations (choosing tuples from within an object), and filter queries which choose objects from a document set (including link traversals.) It is the last type which is most interesting in distributed processing of HyperFile queries, as set and basic selection operations only operate on one or two objects.

2.1. Filter Queries

Filtering queries start with a set of objects, and produce a new set which may contain some of the items in the original as well as items which are reachable from those in the original set. There are two types of operations which happen in a query:

- An object may be tested to see if any of its tuples match particular criteria (example, does the item contain object code?)
- A pointer may be followed; the item pointed to will become one of those being processed.

A sample query, to find all objects in the set S (as shown in Figure 2) which were written by *Joe Programmer*, is:

S | (String, "Author", "Joe Programmer") → T

This takes the objects pointed to by S (*L*, *M*, and *N*); checks to see if they have a tuple of type **String** with the key **Author** and data **Joe Programmer**; and puts the resulting items (only *M* in the example) into the set T. We can also write a query to find the programs in S *and in the routines they call* which are written by Joe:

S | (Pointer, "Called Routine", ?X) | ↑↑X |
(String, "Author", "Joe Programmer") → T

In this case we again start with the items pointed to by S. Tuples which contain the key **Called Routine** are selected, and the value of the pointer (for example, the pointer to *C*) is placed in the variable X (using the ?X operator.) Note that X is a set-valued variable, and thus can contain many references.[2] In the next part of the query, the values placed in each X are dereferenced using the operator ↑↑X.[3] This adds *C* to the set of "possible results" (which becomes {*M, N, L*} ∪ {*C*}.) The last part of the query checks for the presence of the author **Joe Programmer** in the items. The objects which meet this criterion (*M* and *C*) are placed in the result set T, which can be used in further queries just like the set S. Note that the key **Called Routine** is used to select a particular category of pointer; we could use a wild card (?) in place of the key **Called Routine** if we wished to follow all pointers (such as the **Library** pointer.)

Note that we do not handle backward chaining, such as *find all routines that call this one*. If such queries are of interest, the application can explicitly incorporate back

[2] A variable set with ?X can also be used to compare different tuples within a document; for example to find routines that are "Maintained by" one of the "Author"s. We will not describe this in detail as it is unrelated to *distributed* query processing; a complete description is in[3].

[3] The ↑↑X operator keeps the pointing object as well as the item referenced. There is also an operator ↑X which keeps only the referenced object.

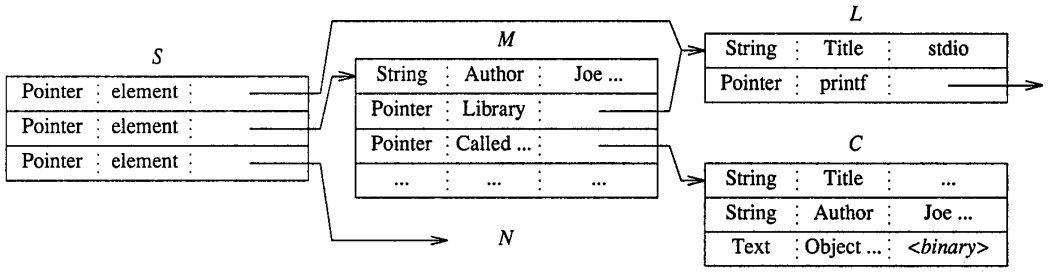

Figure 2: Set of routines from a Software Engineering Application.

pointers in the objects. This fits with our policy of providing a low-level service on which applications are built.

Iteration is also provided, in case we wish to traverse the graph created by the pointers. The iteration can occur a fixed number of times, or can continue indefinitely (to find a transitive closure of the reference graph.) Expanding the "called routine" query to check the transitive closure of the called routines in S would be done as follows:

```
S [ | (Pointer, "Called Routine", ?X) | ↑↑X ]* |
    (String, "Author", "Joe Programmer") → T
```

Replacing the $]^*$ with $]^3$ would cause the iteration to terminate after three levels of pointers have been traversed. The meaning of $[\langle query\ part\rangle]^k$ is to repeat *<query part>* k times, as if the loop was unrolled and executed straight through.

This last query illustrates the main goal of our query interface. In a conventional hypertext system, the above query would require repeated user actions (manual navigation.) A conventional file system would also require repeated interactions. HyperFile performs the full query with a single request to the server.

Like our interface, G+[6] provides for graph based transitive-closure queries. However, computing some G^+ queries can be NP-hard[12]. We have tried to keep our interface simple, so that all queries will be computationally feasible. Our filter queries provide for the common queries we expect to see in document applications. As a matter of fact, we interviewed a number of potential users to learn what requirements they had for a back-end "document" server. These users included hardware designers, programmers, hypertext users, and users of other document retrieval systems[3]. From our discussions we learned that chained queries (our | operator), pointer dereferencing (↑ and ↑↑) and, of course, selection were very common. We believe that the vast majority of searches in such applications can be easily and succinctly expressed in the language of our interface.

The preceding queries do not illustrate how results are actually provided to the application. HyperFile only recognizes certain simple data types (such as pointers, numbers, and keywords.) The bulk of the data is viewed as a sequence of bits, much like a file in a file system. These are retrieved using the → operator. Note that the HyperFile query interface is an embedded language; the → retrieves the requested field into a variable in the application programming language. For example, a C application program could contain:

```
n = 1;
S | (String, "Author", "Chris Clifton") |
      (String, "Title", →title) → T
   { printf("Title %d: %s0, n++, title) }
```

to display individually all of the titles of documents (neatly numbered) in S written by Chris Clifton. Note that these are exactly the documents in T, which can be used in further queries. The above variable title can be of any type in the programming language of the application.

Note that the above retrieval must be done explicitly; queries which just search for objects of interest will not cause any data to be returned. The majority of queries will be used to construct a set of interesting items. These queries need not send large amounts of data (text, bitmaps, etc.) When the set of items of interest is small enough that the user actually wants to see them, a query is issued to retrieve just the desired fields. In addition, we take advantage of large memories (such as in the Massive Memory Machine project at Princeton[8]) to cache all of the pointers, keywords, and other such search information so that disk access is only required to obtain large items.

Due to space limitations we have not described all the facilities of HyperFile. A more complete description of the data model and query interface is given in[3]. In addition to the distributed server, we have developed facilities for indexing[4]. These support conventional indexes (say for keywords in documents), as well as indexes based on the reachability of an object (to speed up queries such as "Find all documents referenced directly or indirectly by this document that in addition have a given keyword").

Finally, note that the query interface we have described is not intended for end users. Instead, application-specific

interfaces will be used, and the application will compose the HyperFile query. For example, in a programming environment the user may first choose what to search for (variable name, author), and then be provided with three main choices: look in the current module, in all called modules, or in the entire program containing the current module. The application would then use these choices to generate a HyperFile query. We are currently developing a graphical/menu driven interface as one type of HyperFile application.

3. Query Processing

The filtering queries of HyperFile are simple to process in a distributed system. Pointer traversal is handled by sending the query along the pointer; the only data which must be sent across the network are the results of the query. We will first present the processing algorithm ignoring remote pointers, and then show the details of handling remote pointers.

First let us introduce a notation for representing queries. Let a query Q be:

$$Q : S_i \; F_1 F_2 \cdots F_n \rightarrow S_\theta$$

where S_i is the initial set of objects, S_θ is the result set of objects, and each F_i is a filter operation of the form:

F_i : (*type*, *pattern*, *pattern*)	;; Selection of tuples
$\uparrow$*matching_variable*	;; Dereference
$\uparrow\uparrow$*matching_variable*	;; Dereference retaining ;; referencing object
I_j^k	;; Iterator starting at F_j, ;; ending at F_i, and ;; repeating k times.

The *pattern* in the tuple selection filter operation varies depending on the type of the value. It may be a string, a range of numbers, or a matching variable.

Let us look at a sample query: Take all of the items in the set S and choose those which contain the keyword *Distributed*. In addition, follow reference pointers for three levels searching for objects which meet these criteria.

S [| (pointer, "Reference", ?X) | $\uparrow\uparrow$X]^3 |
(keyword, "Distributed", ?) $\rightarrow$ T

In the above query, F_1 = (*pointer*, *Reference*, ?*X*), a selection operation which sets the matching variable X. $F_2 = \uparrow\uparrow X$, a dereference of the matching variable. F_3 is the iterator I_1^3, which starts at F_1 and causes pointers to be followed for up to three levels. The last filter F_4 = (*keyword*, *Distributed*, ?) does simple pattern matching: Any object containing a tuple with type *keyword*, key *Distributed*, and any value for the data field will pass this section. The initial set S_i is S, and T will be bound to the result set S_θ.

Certain temporary information will be associated with each object O which is processed by a query. These are:

O.id	The unique Object id (used to retrieve the object.)
O.next	The index of the next filter F_i to process the object.
O.start	The first filter to process the object. For objects in the initial set S_i this is 1. Objects reached as a result of a dereference will have their *.start* set to the filter following the dereference.
O.iter#	The current iteration of an iterator; this corresponds to the length of the pointer chain used to reach O from the initial set.
O.mvars	A table of bindings of matching variables for the object. This is a function $O.mvars(X) \rightarrow \{values\ for\ X\}$.

3.1. Local Processing

The basic means for processing queries is to create a **working set** W containing objects in the original set S.[4] An object is taken from the set and passed through the query from left to right. At each stage it can pass or fail to pass a filter, and may add new objects to the working set. At each stage the object is processed using the function E:

$$E(F_i, O) \rightarrow \{O_x, \cdots\}, [O]$$

E takes a filter and an object; and returns a (possibly empty) set of objects obtained through dereferencing, and either the initial object (if it passed the filter) or null. The actions of E are determined by the type of the filter F_i:

- If F_i is a selection (pattern matching) operation, such as F_4 in the example query, the return set of dereferenced objects is empty. Each tuple of O is processed as follows: If the type field of the tuple matches the type field of the filter, the key and data fields are checked. If these fields match, the object passes the filter. The pattern can be a variety of things, "Matching" depends on what the pattern is:

  The pattern may be a simple comparison (such as a regular expression for strings, or a range of values for a number). In this case matching involves equivalence of the pattern and the field in the tuple. The meaning of equivalence depends on the type of the field.

  The pattern may be a ?, such as in F_4. This matches anything.

  The pattern may set a matching variable, as in F_1. The ?X adds the field value to the bindings for X (if the tuple otherwise matches.) Formally,

[4] The choice of data structure for the working set determines the search order for the algorithm, for example a queue gives breadth-first search. Work by Sarantos Kapidakis shows that a node-based search (such as a breadth-first search) will give the best results in the average case[10].

$O.mvars(X) = O.mvars(X) \cup \{field\_value\}$.
The field matches regardless of value, as with ?.

A matching variable may be used (as described in footnote 2 on Page 3.) In this case, the field matches if any of the values of the matching variable match the field value, that is $field\_value \in O.mvars(X)$.

To be more precise we will give pseudocode for the E function in the case of a selection filter. The details of pattern matching have been left out, as formalizing them requires a discussion of data types and other concerns which are beyond the scope of this paper.

```
E( (type_pattern, key_pattern, data_pattern), O):
  for each tuple t∈ O
    if t.type = type_pattern and
      t.key matches key_pattern and
      t.data matches data_pattern then
        match = true
        Modify O.mvars if key_pattern or
          data_pattern sets a matching variable.
  if match then
    O.next = O.next + 1
    return {}, O
  else
    return {}, null
```

- F_i can be a dereference (↑ or ↑↑). An example of this is F_2 in the above query (↑↑X). In this case E returns a set of all of the pointer values of X. With ↑↑, O is also returned.

```
E(↑X, O):
  Result_set = {}
  for each x∈ O.mvars(X)
    if x is an object id then
      create an object P for processing
      ;; The following line initializes P.
      P.id = x, P.start = O.next + 1,
      P.next = O.next + 1, P.iter# = O.iter# + 1,
      P.mvars = {}
      Result_set = Result_set ∪ {P}
  if the filter is a ↑↑ then
    O.next = O.next + 1
    return Result_set, O
  else
    return Result_set, null
```

Some of the initialization of P in the above needs explanation. $P.next$ is set to the filter after the dereference. $P.mvars$ starts empty; the set contains no bindings. The use of $P.start$ and $P.iter\#$ will be explained in the next paragraph.

- If F_i is an iterator I_j^k, one of two things can happen. If the object has already passed through the entire body of the iterator, or if it is the result of a k length pointer chain, it continues processing with F_{i+1}. Otherwise processing continues at the beginning of the iterator (F_j). Note that iterators do not process objects repeatedly. Operations in the query interface language are idempotent; passing an object through the same filter many times will not change the result. Iterators instead control how often pointers are followed.
  $O.start$ is used to determine if an object has passed through the entire iterator. If $O.start$ is greater than j, the beginning of the iterator, then O must return to the beginning of the iterator. $O.iter\#$ stores the length of the pointer chain used to reach O. For example, if an object P is reached by dereferencing O, $P.iter\# = O.iter\# + 1$. This is done as part of the dereferencing operation shown in the previous section of pseudocode for E. If $O.iter\# \geq k$, O is the result of a pointer chain of length at least k and is not run back through the iteration.[5]

```
E(I_j^k, O):
  if O.start≤j or O.iter#≥k then
    O.next = O.next + 1
  else
    O.start = j  ;; So that O will pass next time.
    O.next = j
  return {}, O
```

Actual processing occurs by creating a working set and filling it with the objects in S_i. The $.next$ and $.start$ indexes for each of these objects is initialized to 1 (the first filter.) Iteration numbers are also set to 1, and the $.mvars$ bindings are initially empty. Each object is taken from the set, and pushed through the filters (using the E function) until they either reach the end or fail to pass part of the filter. Dereferencing operations may add objects to the set. The query terminates when the set is empty.

To give a short example, let us assume that we have a set S containing an object A. A has a reference pointer to B, B has a pointer to C, and C has a pointer to D. We will run the following query (described at the beginning of this section) on the set S:

S [| (pointer, "Reference", ?X) | ↑↑X]^3 |
(keyword, "Distributed", ?) → T

The object A (the only thing in S) is processed. $A.iter\#$ is initialized to 1. In F_1 the matching variable X is set to the pointer (object id) B. F_2 dereferences this, setting $B.start$ and $B.next$ to 3, and $B.iter\#$ to $A.iter\# + 1$, or 2. The initialized B is then added to the set W. Next A continues processing with F_4, which checks for a keyword *distributed* and adds A to T if the keyword is found. Then B is removed from the set, and starts processing at the iterator $F_3 = I_1^3$ (as $B.next = 3$.) Since $B.start > 1$ and $B.iter\# < 3$ we realize B is new to the iterator and the result of a short chain of pointers, so B goes to F_1 (with $B.start = 1$.) Here X is set to C. In F_2 X is dereferenced; C is initialized with

[5] $O.iter\# \geq k$ is not tested if $k = *$. * may be thought of as ∞.

$C.start = C.next = 3$ and $C.iter\# = B.iter\# + 1 = 3$ then placed in W. Next B reaches F_3, but this time $B.start \leq 1$ so it continues processing with F_4. When C begins processing (at F_3) $C.iter\# \geq 3$ and C exits the iteration (continuing with F_4.) Thus the query terminates before examining D (which is 4 levels deep.)

So far we have assumed that iterators are not nested. We do not expect nesting to be common, but it is handled with a slight extension to the above algorithms. The iteration number associated with an object O ($O.iter\#$) is actually a stack of iteration numbers. Where $O.iter\#$ is used in the above algorithms, we actually use the topmost iteration number, which corresponds to the innermost iterator. When a dereference occurs, the new object is initialized by copying the stack, and incrementing only the top iteration number.

Queries which cover the transitive closure of a graph of pointers (queries which contain an iterator $[\langle query\ part \rangle]^*$) pose a potential problem: cycles in the graph of pointers could cause cycles in the processing, preventing termination. This is handled by marking objects as they are processed (actually, noting the object id in a table of used items); if a marked object is found in the working set it is ignored.

However, there is one important subtlety. Consider a query $Q = S_i\ F_1 F_2 F_3 F_4\ S_\theta$. Say a particular object O is in the initial set S_i, but fails to make it through filter F_1. Some other object containing a reference to O makes it through F_1, and in F_2 (a dereferencing filter) the pointer to O is dereferenced. Now we must realize that even though O was seen earlier (at F_1), it still needs to be processed starting at F_3. Thus, our mark table will record not only the identifiers of objects seen by a query, but also where in the query they were seen. In particular, *mark_table*(*object_id*) will store a set of filter numbers. In our example, after processing O at F_1, *mark_table*(O) = {1}. After O is processed at F_3, *mark_table*(O) = {1, 3}. Figure 3 gives the complete query processing algorithm.

Note that there is no *global state* to be maintained between processing of each object in the set other than that in the work set W and the *mark_table*. In fact, the matching variable table $O.mvar$ and "next filter" $O.next$ are only needed while the object is being processed; $O.mvar$ always starts as {} and in all cases $O.next$ is initially equal to $O.start$. The only state which *must* be maintained in W are the object id, iteration number and starting point in the query. This eases the task of parallel processing; to process an object in the set all that must be known is the original query Q, the information in the object O and the *mark_table*. We will see that the *mark_table* can be maintained locally by each site, thus requiring very little *distributed* information.

```
For each object_id x ∈ S_i do ;; Initialize W from S_i.
  create an object O for processing.
  O.id = x, O.start = 1, O.next = 1,
    O.iter# = 1, O.mvars = {}
  append O to W.

While not empty(W) do
  O = head(W);;remove O from the set
  If O.start ∉ mark_table(O.id) then
    While not null(O) and O.next ≤ n do
      mark_table(O.id) =
        mark_table(O.id) ∪ {O.next}
      s, O = E(F_O.next, O)
      W = W ∪ s ;; add all dereferences to the set.
    If not null(O) then
      S_θ = S_θ ∪ {O} ;; add O to the result set
```

Figure 3: Query Processing Algorithm

3.2. Processing Remote Pointers

The basic idea behind processing a reference to a remote site as part of a query is to send the query, not the data. The remote machine processes the query, and returns any results to the originating site of the query. We expect objects in our system to be long relative to the size of a query, so sending the query results in a considerable savings in communication cost over sending the unprocessed objects to the originating site. In addition, processing can continue at the originating site, taking advantage of the parallelism inherent in a distributed system.

Each site keeps a local context for queries it is processing. This context is a set of queries $\{Q_1, Q_2, \cdots\}$ where for each Q_i we have:

Q.id	An identifier for the query (assigned by the originating site.) Combined with *Q.originator*, this forms a globally unique identifier for the query.
Q.originator	The site at which the query was issued.
Q.body	The body ($F_1, F_2, \ldots, F_n$) of the query.
Q.size	The length n (number of F_i) of the query.
Q.mark_table	The set of objects already processed (the *mark_table* described previously.)
Q.W	The working set for this query.
Q.result	The set of results of the query.

A query is processed as follows:

- The originating site sets up a context Q for the query.
- The algorithm of Figure 3 is run, with the context Q used for the working set W, filters F_i, *mark_table*, and result set S_θ.

When the E function returns a set s containing a reference to an object O at a remote site R, that object is not added to the working set $Q.W$. Instead the query and reference are sent to the site R. Specifically the message includes *Q.id*, *Q.originator*, *Q.body*, and *Q.size* from the query context, and *O.id*, *O.start*, and *O.iter#* from the object being dereferenced.

When site R receives the message, it tests if *Q.id@Q.originator* is already in its set of query contexts. If not, Q is added to the local query context, with *Q.result*, *Q.mark_table*, and $Q.W$ set to $\{\}$. Then O is added to $Q.W$, with *O.next* set to *O.start* and *O.mvars* set to $\{\}$. If the algorithm of Figure 3 is not already running (that is, O is the only object in $Q.W$) it is started. Upon termination of the algorithm, *Q.result* is sent to *Q.originator*, and *Q.result* is reset to $\{\}$.

Note that after a site has emptied $Q.W$ and sent results to *Q.originator*, another dereference message for Q may arrive. Since the context Q is still in place, the "setup cost" associated with the query is only required once at each involved site. The context Q is discarded only on global termination of the query.

Note that all sites run an identical algorithm. The message setup time for a remote dereference is minimal: *Q.id*, *Q.originator*, *Q.body*, and *Q.size* are fixed for each query; and *O.id*, *O.start*, and *O.iter#* must be determined for both local and remote dereferences. Thus the cost of processing a distributed reference (at the "pointing" site) is just the cost of sending a message.

The originating site will also receive result messages. Since results are sent directly to *Q.originator*, no intermediate site need be involved in handling the results. Result messages are tagged with *Q.id* so that the originating site can place them in the proper result set. There are two types of results:

- Object identifiers for objects that have passed all of the filters. These are put into the result set S_0 (*Q.result*) at the originating site. Further queries may use this set as a starting point (initial set S_i.)
- Tuple values returned using the $\rightarrow$ operator (such as the example on Page 4.) These are sent to the originating site with a tag noting which $\rightarrow$ they belong to, so they can be bound to the proper variable in the application (*title* in the example.)

Cycle detection and marking are handled locally at each site. The information kept in *Q.mark_table* at each site refers only to objects processed at that site. If a site R has already processed an object O, and later another pointer to O is dereferenced, a message will be sent to R requesting that O be processed. Object O will be placed in the set W at R, but when it is removed from the set the "already processed" mark will be found in *Q.mark_table* and O will be ignored.

This method does allow messages requesting that already processed objects be processed. Eliminating the extra messages (the second and later ones asking that O be processed) would require a global mark table. We believe the cost in communications and complexity of such a global table would outweigh the cost of the extra messages generated by the algorithm we use.

4. Other Issues

We have described the basic distributed query processing mechanism. Some details have been left out; we briefly describe query termination and the naming of objects in a distributed environment here.

With only a single site, a query terminates when its working set is empty. With multiple sites, however, all of the working sets must be empty. Determining when this has happened is an instance of the *Distributed Termination Problem*[7], which has been the subject of considerable research. A number of algorithms to solve this problem have been developed. One that is particularly appropriate to HyperFile is the *weighted messages algorithm*[9, 13], which has been implemented in our prototype.

There are a number of ways to map an *object id* to a specific object at a particular site. Name servers[1] can add to the cost of dereferencing a pointer, particularly if the name server is at a remote site. The obvious alternative of including the host site as part of the pointer seriously increases the cost of moving an object, as all pointers to the object must be updated if it changes sites. We use a variant of the method of R*[11] which includes the *birth site* and the *presumed* current site of an object in the name. The birth site is the final arbiter of the actual location of the object.

5. Experiments

We have implemented this algorithm in a prototype HyperFile server, distributed over a network of IBM PC/RTs connected by an ethernet. The RTs run Berkeley 4.3 UNIX; UDP and TCP/IP are used for inter-process communication. Each machine has a single server. This is a main memory database (as described in Section 2.1); although large objects are stored on disk none of our test queries required disk access. The implementation is not particularly efficient; it is built using an object-oriented programming system (Eiffel) and we have concentrated on extensibility rather than speed. An optimized system would significantly decrease the times we present. Our experimental client read a query from a script, submitted it to HyperFile, received the result, and then went on to the next query in the script. The client ran at a separate machine from any of the servers.

We ran some performance tests on this system. The goal of our experiments was to understand the tradeoffs involved in handling remote pointers:

- Overhead: Extra work is involved in sending messages and processing results from remote sites. Do queries involving remote pointers give unacceptable response time?
- Potential parallelism: Response time may improve when remote processing is started while local processing continues.
- Problems with delays: If the last object to be processed locally contains a remote pointer, the entire system may be idle while that message is in transit.

Note that we do not yet have a reasonable "competitor" algorithm or system to compare our performance with. Performing similar queries in a distributed file system would require searching entire files; this in effect results in sending all data to a central site. At best this uses a single message for each file, the *worst-case* for HyperFile requires a message for each object. Our messages send only the query (about 40 bytes for the experiments presented here) versus potentially huge messages required to send a complete file. Hypertext systems require manually "browsing" through the data, and are not commonly distributed. Neither would be an interesting comparison.

We constructed synthetic data to use in our experiments. This allowed us to "parameterize" our tests, so we could load the system in various ways and study the results. In particular, each object searched as part of our test queries contained the following:

- Five **search key** tuples; one guaranteed to be unique to that object, one found in all objects, and three which were chosen from a space of 10, 100, and 1000 possible values respectively. Changing the tuple and value searched for allowed us to vary the number of items found by a query. For example, searching for a given key in the *unique* tuple would return one object.
- One **chain** pointer, which gave a linked list of all the items. In tests with more than a single machine, these pointers were always to a remote machine. This gives the maximum *delay* time; all servers are idle while each message is in transit.
- Fourteen **random** pointers. These each pointed to a randomly chosen object. They were divided into 7 types, with two pointers of each type. The probability of a pointer being to a local object varied from .05 to .95 depending on the type. For example, the two pointers of the **Rand.05** type were almost always to a remote object. A query following the Rand.05 pointers would have high message cost. However, since there were two such pointers in each object (very likely to different machines) the query would "branch out", yielding some parallelism and reduced delays.
- **Tree** pointers which formed a spanning tree of the objects, such that the root of the tree had a single remote pointer to all other machines, and each of these was the root of a local spanning tree. This gives high parallelism with low message cost.

We ran tests with these items divided evenly among three machines and among nine machines. The pointers were constructed such that the desired properties (likelihood of a pointer being remote, etc.) were the same in both cases; i.e., the graph formed by the pointers in these objects was identical regardless of the number of machines. We also ran the tests with all items on a single machine. This gave a base case with which to compare the cost of handling remote pointers.

Each query traversed the transitive closure of the graph formed by a particular type of pointer, and looked for a given search key within each item in the transitive closure. For example, the query

Root [| (Pointer, "Tree", ?X) | ↑↑X]* |
(Rand10p, 5, ?) → T

would traverse the *tree* structured graph (splitting immediately to each machine, and then tracing pointers locally on that machine.) Each object would be checked to see if it had a **Rand10p** tuple with a key of **5** (Since each item had a single Rand10p tuple, with its key value randomly distributed from 1 to 10, we would expect the result to contain about 10% of the items in the tree.)

From our experiments we deduced a few basic times. Local processing of a single object took approximately 8 milliseconds, plus another 20 milliseconds to add the object to the result set (if necessary.) The added time to process a remote pointer was roughly 50 milliseconds (including constructing the message, system calls for sending and receiving, and transmission delay.) About 50 milliseconds was also required for each remote result message. Of course, remote pointers may allow parallel processing of queries, so the extra time to process a remote pointer does not necessarily translate into an equivalent increase in client response time.

Perhaps more interesting than the above numbers is the actual query response time. We tried a number of cases, all based on the transitive closure query shown above. The graph structure was varied with each test; we tried extreme cases (such as **Chain**, giving maximum delay; or **Tree**, giving high parallelism at low message cost) as well as the randomly created graphs with varying locality of reference. We also tried varying the quantity of items returned (by changing the tuple in the search key.) For each test we timed 100 queries which followed the same pointers and looked for the same *type* of search key tuple, but randomly varied the key searched for (so the 100 queries were comparable, but not identical.) This time was the actual response time (wall clock) at the client.

There were 270 objects involved in the queries for which we report results. (Note that the total database was larger; however only 270 objects were looked at by our test queries.) As the algorithm is linear we expect using a different number of items in the query would result in a linear change in the response time. We did construct a

data set with half the number of items; this didn't quite cut the query time in half. This is as we would expect (since there is some constant overhead associated with the query, regardless of size.) Presenting more experiments with varied data set sizes would tell little of interest; our primary concern is how remote pointers affect performance.

Running the query shown above (a transitive closure over 270 items, with approximately 27 in the result set) took 2.7 seconds when all the objects were at a single site, when following either tree or chain pointers.

In the worst case delay scenario (following *chain* pointers) in the distributed case (on either three or nine machines) the query took 15 seconds. The delay and message cost of such a query is high, however pointers with such a structure can probably be avoided in practice. When we instead followed *tree* pointers a query averaged 1.5 seconds using three machines, and 1 second using nine machines. We obviously gain from parallelism in this query; times are significantly less than for a single site.

The above two cases are extremes. To study "normal" situations we ran tests on the randomly constructed pointers. Although still synthetic data, they are probably more representative of real situations. The results of these tests are graphed in Figure 4. Each data point represents a test using the graph formed by the pointers with the given probability (*x axis*) of being local (two such pointers per object.) The cases at the far right of the graph generate fewer messages, however they also are less likely to make full use of the available parallelism. The cases at the far left generate too much message traffic for our system; although parallelism is increased, much of the time is spent receiving and sending messages rather than processing queries. We see that the system operates best with at least 80% local references. We can also see that with more machines we are more capable of handling a higher percentage of remote references. This is good, as a more highly fragmented database will probably have more remote references.

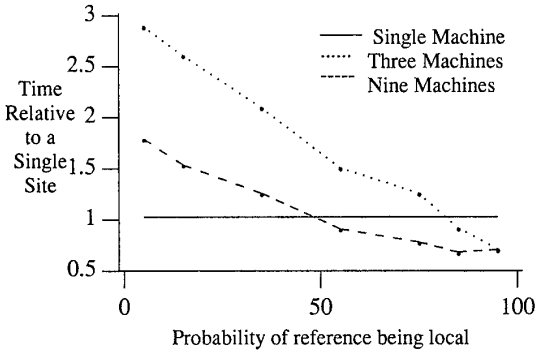

Figure 4: Query speedup with increasing probability of local references.

Another interesting result concerns the number of items returned by a query. Increasing the number of items returned significantly increases the query processing time. Given two queries that follow the same pointers, a highly selective query may be faster in the distributed case, while a less selective query may run faster when the entire database is on a single server. For example, the case in Figure 4 where 95% of the pointers are local takes an average 1.1 seconds when run on three or nine machines, and 1.5 seconds when run at a single site. Note that this is returning an average 10% of the items in the transitive closure. If we instead select all of the items (using a key which is found in all of the objects) the single site time jumps to 5.1 seconds. For three and nine sites we have 6.4 and 5.7 seconds. Sending results is expensive in our system; we would have to make changes if queries with low selectivity are frequent. We expect this will not be the case, as the goal of most queries is to find a *few* interesting objects.

There is a straightforward modification that would help this problem. In the case of queries which only construct a new set (as opposed to returning specific fields from objects) the result could be left as a "distributed set". Each server would send back the number of local result items, rather than pointers to the items themselves. If this number is large, the user will probably want to further restrict the results using a query rather than look at the returned items. The portion of this set at each site would be used to initialize the working set at that site for the new query. This method would probably be employed only when the size of the results exceeded some threshold.

Given that the goal of this system is efficient *distributed* query processing as opposed to *parallel* processing, the results are reasonable. In all but extreme cases, remote pointers do not significantly increase response time. The cost of processing messages and the transmission delay are substantially offset by the gains in parallel processing. We see that the cost of distribution is low (with respect to response time, normally the most important measure to the user of an interactive system.)

6. Conclusions and Further Work

We have described HyperFile, a back-end data service for heterogeneous applications. It provides a query language that permits searches based on properties of the stored objects, as well as by following pointers contained in the objects. We believe that the query language is powerful enough so that many common queries in applications such as document processing can be answered with a single request to HyperFile. Yet, HyperFile is simple enough so that it can be efficiently and easily implemented in a distributed environment.

This paper has focused on the algorithms for distributed filter queries. With the resulting algorithms, a search on a distributed network of object causes the query and not the objects to move along the links. We also discussed HyperFile's naming strategy and query termination algorithm, as well as experiences with a prototype.

Although we have covered the case of a distributed HyperFile server, it is important to note that our algorithms are also applicable to a shared memory, multiprocessor server. In this case all available processors can share the same general query information, mark table, and working set. Each processor must have space for local information, such as matching variables, while it is processing a particular document. Given this, each processor independently runs the algorithm of Section 3.1. Termination requires that the set be empty, *and* that no processors are still working on the query. Note that this is similar to processing of the Linda language[2]. Also notice that it is not necessary to have a strict locking mechanism to prevent two processors from working on the same document. Duplicate processing may create some duplicate answers, but not incorrect ones (due to the set-based nature of the result.)

We are currently working on a simple driving application. This application is a simple hypertext system. It allows conventional hypertext browsing operations. In addition, it lets the user pose HyperFile style queries that will be forwarded to HyperFile for processing. We believe this may address the "lost in hyperspace" problem that arises in large hypermedia databases. This problem refers to the inability of users to retrieve a document because they cannot manually construct the right path to it. With HyperFile, the user is able to pose powerful queries to automate the search for relevant documents.

Acknowledgements

Some of the ideas described in this paper were initially developed at Xerox P.A.R.C. in discussions with Robert Hagmann, Jack Kent, and Derek Oppen. We would like to acknowledge their contribution.

References

1. Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder, "Grapevine: An Exercise in Distributed Computing," *Communications* **254**(4) pp. 260-274 ACM, (April 1982).
2. Nicholas Carriero and David Gelernter, "The S/Net's Linda Kernel," *Transactions on Computer Systems* **4**(2) pp. 110-129 ACM, (May 1986).
3. Chris Clifton, Hector Garcia-Molina, and Robert Hagmann, "The Design of a Document Database," pp. 125-134 in *Proceedings of the Conference on Document Processing Systems*, ACM, Santa Fe, New Mexico (December 5-9, 1988).
4. Chris Clifton and Hector Garcia-Molina, "Indexing in a Hypertext Database," pp. 36-49 in *Proceedings of the 1990 International Conference on Very Large Databases*, VLDB, Brisbane, Australia (August 13-16 1990).
5. Jeff Conklin, "Hypertext: An Introduction and Survey," *Computer* **20**(9) pp. 17-41 IEEE, (September 1987).
6. Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood, "A Graphical Query Language Supporting Recursion," pp. 323-330 in *Proceedings of SIGMOD '87*, ACM, San Francisco, CA (May 27-29, 1987). Also SIGMOD Record Vol. 16 #3, December 1987.
7. Nissim Francez, "Distributed Termination," *Transactions on Programming Languages and Systems* **2**(1) pp. 42-55 ACM, (January 1980).
8. H. Garcia-Molina, R. J. Lipton, and J. Valdes, "A Massive Memory Machine," *Transactions on Computers* **C-33**(5) pp. 391-399 IEEE, (May 1984).
9. Shing-Tsaan Huang, "Detecting Termination of Distributed Computations by External Agents," pp. 79-84 in *Proceedings of the 9th International Conference on Distributed Computing Systems*, IEEE, Newport Beach, CA (June 5-9, 1989).
10. Sarantos Kapidakis, "Average-Case Analysis of Graph-Searching Algorithms," Ph. D. Thesis, Princeton University, Princeton, NJ (October 1990).
11. Bruce Lindsay, "Object Naming and Catalog Management for a Distributed Database Manager," pp. 31-40 in *Proceedings of the 2nd International Conference on Distributed Computing Systems*, IEEE, Paris (April 8-10, 1981).
12. Alberto O. Mendelzon and Peter T. Wood, "Finding Regular Simple Paths in Graph Databases," pp. 185-193 in *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, VLDB, Amsterdam (Aug. 22-25, 1989).
13. Kazuaki Rokusawa, Nobuyuki Ichiyoshi, Takashi Chikayama, and Hiroshi Nakashima, "An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems," pp. 18-22 in *Proceedings of the 1988 International Conference on Parallel Processing*, (August 15-19, 1988).
14. Gio Wiederhold, *File Organization for Database Design*, McGraw-Hill, New York(1987), p. 107.