Distributed Processing of Filtering Queries in HyperFile

Chris Clifton
Hector Garcia-Molina

CS-TR-295-90

November 14, 1990

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.

Distributed Processing of Filtering Queries in HyperFile[†]

Chris Clifton[‡]
Hector Garcia-Molina
Princeton University

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

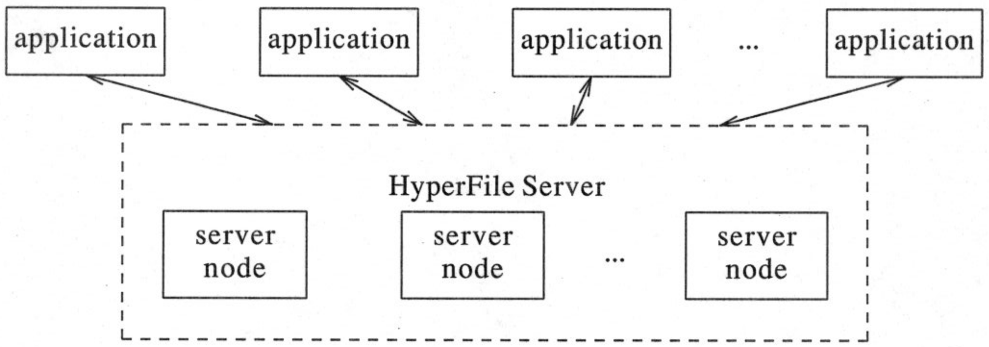

Figure 1: HyperFile *as a back-end service.*

[†] 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.

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 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 of there 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. These will be surveyed in Section 3, after we have given a more detailed overview of HyperFile. 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.

2. The Query Language

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''.

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 4. (The internal structure of pointers is discussed in Section 5.2.)

It is possible for an application to 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 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*[Wied87a]. 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*[Conk87a]. 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 language based on *document sets* and *filtering*. Items returned by a query are determined both 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 for *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*, the program object shown above, 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") $\rightarrow$ 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) | $\uparrow\uparrow$X | (String, "Author", "Joe Programmer") $\rightarrow$ 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. In the next part of the query, the values placed in each X are dereferenced

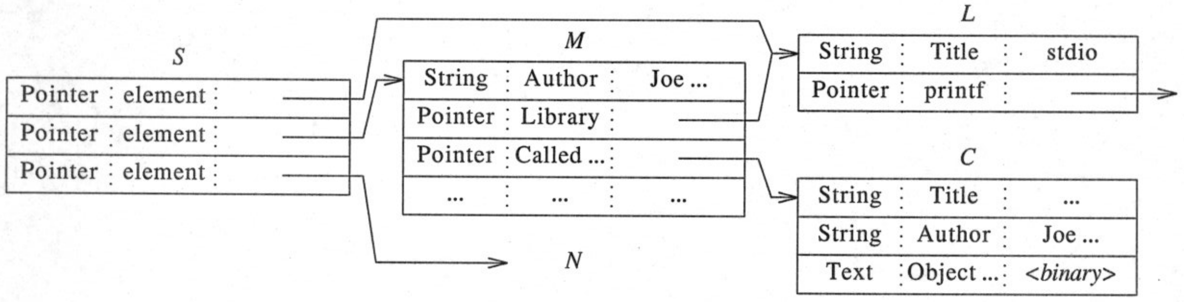

Figure 2: Set of routines from a Software Engineering Application.

using the operator ↑↑X.[1] 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.)

Set variables, such as X in the above example, take on a different set of values for each object. This allows comparison of tuple values within a document, for example choosing programs which are being maintained by their author:

S | (String, "Author", ?X) | (String, "Maintained By", X) → T

In the portion of the query "Author", ?X; X becomes a set of all of the *Authors* of the object, and later these are compared against the values of *Maintained By* tuples. If any of these matches a value in X the expression evaluates true and the program "passes" the query.

Comparisons *between* objects are not part of queries, but are done by applications with the results of queries. An example would be gathering multiple papers on the same subject (without a particular subject in mind), the obvious solution of choosing objects with the same keywords cannot be done as part of a single HyperFile query. We expect that the desire for such queries will be rare in applications supported by HyperFile. Providing for such queries would increase the complexity of query processing considerably. Applications can answer such questions using a few queries, along with some local processing by the application. This will prove useful for distributed query processing.

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 [*<query part>*$]^k$ is to repeat *<query part>* k times, as if the loop were unrolled and executed straight through.

This last query illustrates the main goal of our query language. 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 language, G^+[Cruz87a] provides for graph based transitive-closure queries. However, computing some G^+ queries can be NP-hard[Mend89a]. We have tried to keep our language 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[Clif88a]. 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 our language.

[1] 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.

The preceding queries do not illustrate how results are actually provided to the application. Values of fields in a tuple must be retrieved explicitly using the → operator. The HyperFile query language is used as an embedded language; viewing actual tuple values is done by placing the values in variables in the application programming language. For example, the application program could contain

```
T | (Text, "Description", →descr)
  { show_description (descr) }
```

to display individually all of the descriptions of the programs in the set T using the procedure **show_description** written in the host language.

The above variable descr can be of any type in the applications programming language. HyperFile sees this data only as a string of bits.

Note that the above retrieval must be done explicitly; queries which simply 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[Garc84a]) to cache all of the pointers, keywords, and other such search information so that disk access is only required to obtain large items.

The translation from string of bits to a data structure in the application is analogous to that which occurs when reading and writing files in a file system. This can be used to modify applications for use with HyperFile with a minimum of effort. Instead of storing data in a file it is stored in a HyperFile tuple. The data structures and organization of the application need not be changed. The application can then add tuples with properties to be used in queries, even though this may duplicate information already contained in the "file" tuple.[2] For example a TEX document could be placed as the data field of a single tuple (text, "TeX", *<TeX source>*). Applications would treat this data field much like a file for use by the TEX processor. Properties such as the author and title of the document would be placed in other tuples in the same object to be used for queries.

We have not discussed queries which update the database. Due to the nature of the data, most updates will involve a single object. Examples would be editing a document, or running an enhancement program on a picture. We expect that updates which affect a number of objects at once to be rare. Therefore HyperFile provides for modification, addition, and deletion of single objects. Wider ranging updates may be built as applications. For example, installation of a new compiler may require all object code for a machine to be recompiled. An application would issue a query to construct a set of all objects containing object code for that machine. Each object would then be retrieved, the code recompiled, and the object code tuple replaced.

Due to space limitations we have not described all the facilities of HyperFile. A more complete description of the data model and query language is given in[Clif88a]. In addition to the distributed server, we have developed facilities for indexing[Clif90a]. 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 language we have described is not intended for end users. Instead, application-specific interfaces will be used, and the application will compose the HyperFile query.

[2] An alternative would be to provide a function to extract the property from the "file" tuple automatically. This has two problems: It increases query time, and requires running application code at the server. The problem of keeping the duplicated information consistent becomes a problem of keeping the extracting function current. Security is also a concern; the function must not be allowed to affect the server or database.

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.

The following section discusses where we stand in relation to existing systems. Section 4 shows how filter queries are processed. Other issues in a distributed environment are discussed in Section 5. Finally, in Section 6 we will discuss some results from tests done in a system using servers implementing this algorithm.

3. Comparison With Other Systems

In this section we briefly compare HyperFile to some other data storage systems. While many of these could be used instead of HyperFile as back-end storage facilities, we will argue that for document processing they do not strike the right balance between off loading application-dependent data processing to the front-end and performing the purely data search functions at the back-end.

3.1. File Systems

HyperFile is probably most similar to a file system, particularly one with *self-describing data records*[Wied87a]. In these systems records of a file contain tags stating what information is contained in the record, as opposed to either a heavily structured file (where each record contains the same type of information) or totally unstructured files.

Most electronic documents are currently stored in file systems, rather than databases. This is because of the flexibility allowed in the contents of a file. This freedom is necessary for documents, due to the combination of text, drawings, and other media. Many other applications require this as well; databases for software engineering systems, CAD tools, and other such applications are often custom-designed or built on file systems. In addition, most documents, although structured, are not *rigidly* structured; variations are acceptable when necessary.

File systems allow this flexibility, but provide little structure in places where it is desired. Items can be grouped in directories, and often hierarchical structure of the directories is allowed, but references and other pointers which are a part of many objects are not recognized by file systems. As discussed in the Introduction, file systems are inefficient for search and retrieval. In a large distributed system, this problem is magnified. HyperFile can be viewed as a powerful file server: It provides for storage of unstructured data, but allows much more powerful queries based on the properties of files (objects) and their relation to other objects.

3.2. CODASYL Systems

HyperFile is similar to CODASYL[DBTG74a] in that they both provide objects and pointers. Work has been done on creating a distributed CODASYL[Germ81a]; this requires extensions to the CODASYL standard. However, a major difference between HyperFile and CODASYL is that CODASYL pointers must be used in a very structured way, as parts of predefined *sets*. The database schema determines where pointers are allowed and what they may point to. All items in a set are of the same type. HyperFile does not place such restrictions on the structure of data. Pointers may be used freely, wherever the user or application desires. Although there are difficulties in providing this flexibility (for example, indexing becomes a much more difficult problem)[Clif90a], we feel that the tradeoff is worthwhile for our applications.

Another difference is the query language. The CODASYL query language only allows searches over a fixed set; the scope of a search can be determined from the database schema. We allow queries which arbitrarily follow pointers. This allows for fewer server-application interactions. For a query which covers the transitive closure of a portion of the graph of pointers, CODASYL may require many such interactions, where HyperFile would require only one.

3.3. Information Retrieval Systems

HyperFile is also similar to conventional information retrieval systems[Salt83a] such as those for library applications. These systems allow filter queries similar to ours (e.g., find books with a given title), and indeed, our language was inspired by them. However, conventional information retrieval systems do not understand pointers. The ability to follow pointers within a query is essential to us, especially to support hypertext applications. Also, information retrieval systems typically do not support non-text data.

We view information retrieval systems as likely candidates for HyperFile applications. Ideas from these systems, combined with hypertext methods, can be used to form a general interface to a HyperFile database. Information retrieval research into automatic indexing[Salt88a] and natural language[Crof87a] can also be used to generate properties for textual objects.

3.4. Relational Systems

Relational systems provide a regular structure for data. HyperFile supports data which does not fit into a regular structure. Although work has been done on placing text items in a relational database[Ston83a, Smit86a], creating a relational database which can support a variety of heterogeneous types of data is difficult. Conventional relational systems do not support pointers and this is a serious shortcoming for us. Steps have been taken to address some of these problems in ''advanced'' relational systems (pointers, flexible data types, etc.), but we address these below.

3.5. Advanced Database Systems

Advanced database systems (object oriented[Maie86a, Woel86a, Wein88a], extended relational[Ston86a, Schw86a, Dada86a]) provide many of the facilities of HyperFile (objects, pointers, queries), but also provide a lot more (like a full programming language or an inferencing engine). We feel that these systems may provide *too* much for a back-end document data server.

In particular, an advanced database system could open the door for doing much of the application processing at the back-end. We feel that this can create unreasonable processing loads at the shared server. Of course, one can restrict the general interface to allow only certain queries and a simple model for objects. But if this is the case, there is no need to have a full and complex object-oriented schema and programming interface at the back-end! Restricting the interface, as in HyperFile, makes it much easier to perform efficiently the queries that are allowed, and much easier to distribute the back-end service.

3.6. G^+

G^+ is a graph query language developed at the University of Toronto[Cruz87a]. It has common goals with HyperFile. Their language, although more general, poses computational complexity problems (as mentioned in Section 2.1.) This defeats our goal of providing a simple and efficient back-end data storage service.

4. 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
      ↑matching_variable            ;; Dereference
      ↑↑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.

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

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 the result set S_θ is T.

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 $O.mvars(X) \rightarrow \{values\ for\ X\}$.

4.1. Local Processing

The basic means for processing queries is to create a **working set** W containing objects in the original set S.[3] 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:

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

[3] The choice of data structure for the working set will determine the search order for the algorithm, for example a queue will give a 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[Kapi90a].

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. An example of this is F_1. The ?X adds the value of the field of the tuple to the bindings for X (if the other fields match.) More formally, $O.mvars(X) = O.mvars(X) \cup \{field\_value\}$. The field matches regardless of the field value, as with ?.

  A matching variable may be used (as described in the example on Page 5.) 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 $t \in O$
        if $t.type = type\_pattern$ and
          $t.key$ matches $key\_pattern$ and
          $t.data$ matches $data\_pattern$ then
            $match = true$
                ;; Note that $O.mvars$ may have been modified 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 ($\uparrow$ or $\uparrow\uparrow$). An example of this is F_2 in the above query ($\uparrow\uparrow X$). In this case E returns a set of all of the pointer values of X. With $\uparrow\uparrow$, O is also returned.

```
E($\uparrow$X, O):
    $Result\_set = \{\}$
    for each $x \in O.mvars(X)$
        if $x$ is an object id then
            create an object $P$ for processing
            ;; The following line initializes $P$.
            $P.id = x$, $P.start = O.next + 1$, $P.next = O.next + 1$, $P.iter\# = O.iter\# + 1$, $P.mvars = \{\}$
            $Result\_set = Result\_set \cup \{P\}$
    if the filter is a $\uparrow\uparrow$ 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 actually cause objects to be processed repeatedly. Operations in the query 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.[4]

```
    $E(I_j^k, O)$:
        if $O.start \leq j$ or $O.iter\# \geq k$ then
            $O.next = O.next + 1$
        else
            $O.start = j$          ;; So that $O$ will pass the iterator 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 then 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 (see Figure 3.) 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

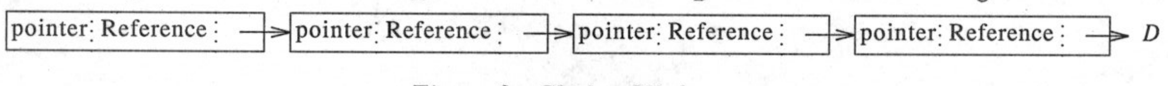

Figure 3: Chain of References.

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

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 then 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 *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*≤1 so it continues processing with F_4. When *C* begins processing (at F_3) *C.iter#*≥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 mapping from an iterator to an iteration number. Where *O.iter#* is used in the above algorithms, we actually use the iteration number corresponding to the iterator being processed. When a dereference occurs, only the innermost iteration number (the pair with the lowest iterator index ≥ *O.next*) is incremented.

Queries which cover the transitive closure of a graph of pointers (queries which contain an iterator [<*query part*>]* 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 4 gives the complete query processing algorithm.

Note that there is no *global state* to be maintained between processing of each object in the set

```
For each object_id $x \in S_i$ do                    ;; Initialize $W$ with objects in $S_i$.
    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 $O.start \notin mark\_table(O.id)$ then
        While not null($O$) and $O.next \leq n$ do
            $mark\_table(O.id) = mark\_table(O.id) \cup \{O.next\}$
            $s, O = E(F_{O.next}, O)$
            $W = W \cup s$                            ;; add all dereferences to the set.
        If not null($O$) then
            $S_\theta = S_\theta \cup \{O\}$          ;;add $O$ to the result set
```

Figure 4: Query Processing Algorithm

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.

4.2. 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 $(F_1, F_{2,...}, F_n)$ 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 in the previous section.)
$Q.W$	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 4 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 4 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 (to be discussed in Section 5.1.)

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_θ (*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 6.) 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 (*descr* 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. Following are pseudo-code descriptions of the query processing operations.

```
Auxiliary Data Structures (at each site) :
    Table of Queries: Query → Marked Documents, Working Queue

Send Message (Query, Point in Query, Reference) :
    send(to Reference.machine, Query, Reference, Point in Query)

Receive Message :
    If message.Query not in Table of Queries:
        Create Working Queue
        Add Table of Queries.message.Query := {}, Working Queue
    add reference, point in query to Working Queue.
    if Process Queries not active, run Process Queries

Process Queries :
    While Working Queue not empty do
        mark document
        if document passes filters, send_result(document_id)
        if any references result, either add to queue or send_message

Send results (document) :
    If query asks for results other than the object id (selection
    query, or query contains the → operator) then
        send(Query.originator, requested portion of document)
    else
        send(Query.originator, document.id)
```

5. Other Issues

We have described the basic distributed query processing mechanism. Some details have been left out; these are described individually in this section.

5.1. Query Termination

With only a single site, a query terminates when its working set becomes 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*[Fran80a], which has been the subject of considerable research.

The problem of distributed termination is to determine when a distributed computation has finished. The computation starts at some *originating site*, and parts of the computation may be sent to remote sites. The computation is complete when no sites have any processing left to do. Note that it is difficult for a site to determine on its own when it is done. Even though it may have nothing left to process locally, another site may later send it a message which will cause it to resume processing.

A number of algorithms to solve this problem have been developed. One that is particularly appropriate to HyperFile is the *weighted messages algorithm*[Huan89a, Roku88a]. This algorithm works as follows:

- The original site starts with some positive weight W.
- Any message (query) sent to another site must include some positive weight w, which is subtracted from the weight of the sending site and added to the weight of the receiving site.
- When a site (other than the original) is done, it sends a message with its remaining weight back to the originating site.
- When the originating site is done, and its weight is back to W, the computation (query) is complete.

Note that the only increase in message traffic is due to the "I'm finished'' messages. In many cases, these can be piggy-backed on the sending of results.

One problem with this method is that it does not very robust. In particular, if a site fails while it still has some weight w, that weight will never be returned to the originating site. Thus the query will never terminate. HyperFile times out if the query seems stalled, and reports partial results to the application. This allows the application (or user) to decide if the missing part of the query is important; in some cases the information found at the available machines may be satisfactory.

More robust distributed termination protocols exist[Lai86a], but they are also more complex and expensive. These could be used in HyperFile in a distributed system where the timeout method is inadequate.

5.2. Naming issues

Pointers form an important part of the database. Each object (set of tuples) has a unique *object id*, which can be used as a pointer to the object. Distributing the data requires a naming scheme so that these pointers can cross machine boundaries. There are two parts to this; ensuring that object identifiers remain unique, and translating pointers (object ids) into the site containing the object. For the latter we need some function $F(\text{Pointer}) \rightarrow \text{Location}$. There are a variety of ways to do this, such as global name servers[Birr82a] or including the name of the host site as part of the pointer. There are a number of tradeoffs in the choice of a naming strategy:

- Storage cost of pointers.
- Execution time and message cost to follow a pointer.
- Costs associated with moving an object. This can be broken down into two types of moves: changing the site at which the ''pointed to'' object is stored, and moving an object containing a

pointer.

Name servers 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^*[Lind81a] which includes the *birth site* of an object in the name.

A HyperFile object id consists of the birth site of the object, and a unique identifier (we use a sequence number) assigned by that site. This solves the problem of maintaining system-wide unique object ids. Each site has a cache which maps this object id into a *presumed site*. This allows most pointer references to proceed directly to the site of the object. If the referenced object has been moved, the message will be forwarded to the birth site. The birth site must always know where any item it has created is located, but no other site must be notified if an object is moved. Cached pointers which are out of date are updated when they are used. The cache at a site *A* does not have to have presumed sites for all pointers from objects at *A*; ''missing'' pointers can be directed to the birth site just like misdirected messages. This simplifies moving an object; only the birth site need be notified. Costs to update pointers from moving either a referencing or referenced object are delayed until the pointers are used.

A slight extension of this can be used to increase reliability and availability through replication. The cached current location of a replicated object would never be the birth site; if this cached location is unavailable the reference would be sent to the birth site. The birth site would be responsible for knowing all copies of the object. Different sites could cache a different current location for an object, any duplicate results are filtered out by the originating site.

6. Experiments

We have implemented this algorithm in a prototype HyperFile server, distributed over a network of IBM PC/RTs connected by an ethernet. The RT's 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 was a simple application that 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. 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 at most 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 wore 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.

When the worst case delay scenario (following *chain* pointers) was tried 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 the 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 5. 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.

It would be reasonable to expect that the single-machine case would be constant. This is not the case. The reason is that the pointers in these tests were created randomly (within the local/remote guidelines), and the transitive closure of a given pointer type was not guaranteed to

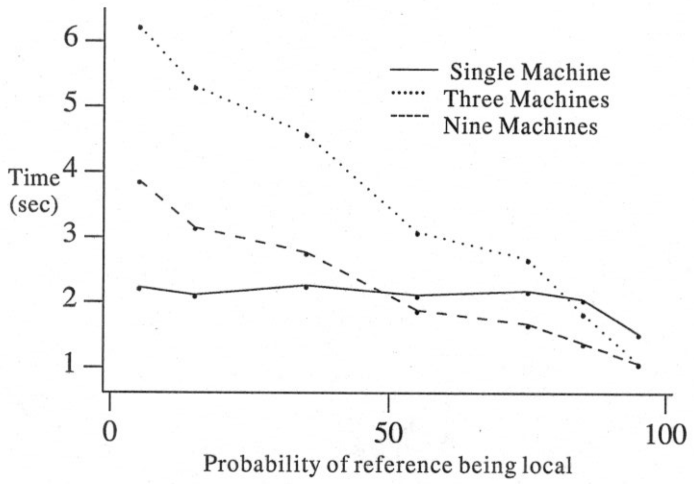

Figure 5: Query time with varied probability of local references.

include 270 objects. The single machine case gives a measure of the number of items actually covered, so it is perhaps more relevant to look at the difference between the dotted or dashed line and the solid line, rather than the absolute times. Based on this we see that the system operates best with around 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.

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 5 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.)

7. 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, multi-processor 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 4.1. 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[Carr86a]. 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.

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 user 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

DBTG74a.
Data Base Task Group, "CODASYL Data Description Language," NBS Handbook 113, National Bureau of Standards, US Department of Commerce, Washington, DC (January 1974).

Birr82a.
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).

Carr86a.
Nicholas Carriero and David Gelernter, "The S/Net's Linda Kernel," *Transactions on Computer Systems* **4**(2) pp. 110-129 ACM, (May 1986).

Clif88a.
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).

Clif90a.
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).

Conk87a.
Jeff Conklin, "Hypertext: An Introduction and Survey," *Computer* **20**(9) pp. 17-41 IEEE, (September 1987).

Crof87a.
W. B. Croft and D. D. Lewis, "An Approach to Natural Language Processing for Document Retrieval," pp. 26-32 in *Proceedings of the 10th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, , New Orleans, LA (June 1987).

Cruz87a.
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.

Dada86a.
P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walch, "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables an Hierarchies,'' pp. 356-364 in *Proceedings of the SIGMOD International Conference on Management of Data*, ACM, Washington, DC (May 28-30, 1986).

Fran80a.
Nissim Francez, ''Distributed Termination,'' *Transactions on Programming Languages and Systems* **2**(1) pp. 42-55 ACM, (January 1980).

Garc84a.
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).

Germ81a.
Frank Germano, Jr., *Automatic Transaction Decomposition in a Distributed CODASYL Prototype System,* UMI Research Press, Ann Arbor, Michigan(1981).

Huan89a.
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).

Kapi90a.
Sarantos Kapidakis, ''Average-Case Analysis of Graph-Searching Algorithms,'' Ph. D. Thesis, Princeton University, Princeton, NJ (October 1990).

Lai86a.
Ten-Hwang Lai, ''Termination Detection for Dynamically Distributed Systems with Non-First-in-first-out Communication,'' *Journal of Parallel and Distributed Computing* **3**(4) pp. 577-599 (December 1986).

Lind81a.
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).

Maie86a.
David Maier, Jacob Stein, Allen Otis, and Alan Purdy, ''Development of an Object Oriented DBMS,'' pp. 472-482 in *Object Oriented Programming Systems, Langauges, and Applications Conference Proceedings*, ACM, Portland, OR (September 9 - October 2, 1986). Also Sigplan notices **21**(11), November 1986.

Mend89a.
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).

Roku88a.
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).

Salt83a.
Gerard Salton and Michael J. McGill, *Introduction to Modern Information Retrieval,* McGraw Hill Book Company, New York(1983).

Salt88a.

Gerard Salton, ''Automatic Text Indexing Using Complex Identifiers,'' pp. 135-144 in *Proceedings of the Conference on Document Processing Systems*, ACM, Santa Fe, New Mexico (December 5-9, 1988).

Schw86a.

P. Schwarz, W. Chang, J. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh, ''Extensibility in the Starburst Database System,'' *Proceedings of the 1986 International Workshop on Object Oriented Database Systems*, pp. 85-92 (September 1986).

Smit86a.

Karen E. Smith and Stanley B. Zdonik, ''Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems,'' pp. 452-465 in *Object Oriented Programming Systems, Langauges, and Applications Conference Proceedings*, ACM, Orlando, Florida (October 4-8, 1986). Also Sigplan notices **22**(12), December 1987.

Ston83a.

M. Stonebraker, A. Stettner, N. Lynn, J. Kalash, and N. Guttman, ''Document Processing in a Relational Database System,'' *Transactions on Office Information Systems* **1**(2) pp. 143-158 ACM, (April 1983).

Ston86a.

M. Stonebraker and L. Rowe, ''The Design of POSTGRES,'' pp. 340-355 in *Proceedings of the SIGMOD International Conference on Management of Data*, ACM, Washington, DC (May 1986).

Wein88a.

Dale Weinreb, Neal Feinberg, Dan Gerson, and Charles Lamb, ''An Object-Oriented Database System to support an Integrated Programming Environment,'' *Data Engineering* **11**(2)IEEE, (June 1988).

Wied87a.

Gio Wiederhold, *File Organization for Database Design,* McGraw-Hill, New York(1987), p. 107.

Woel86a.

D. Woelk, W. Kim, and W. Luther, ''An Object-oriented approach to Multimedia Databases,'' pp. 311-325 in *Proceedings of SIGMOD '86*, ACM (May 1986).