**CERIAS Tech Report 2001-76**
**HyperFile: A Data and Query Model for Documents**
by Christopher Clifton
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

*VLDB Journal*, 4, 45-86 (1995), Ron Sacks-Davis, Editor



# HyperFile: A Data and Query Model for Documents

**Chris Clifton, Hector Garcia-Molina, and David Bloom**

*Received July 1, 1992; revised version received, January 25, 1994; accepted October 7, 1994.*

**Abstract.** Non-quantitative information such as documents and pictures pose interesting new problems in the database world. Traditional data models and query languages do not provide appropriate support for this information. Such data are typically stored in file systems, which do not provide the security, integrity, or query features of database management systems. The hypertext model has emerged as a good interface to this information; however, *finding* information using hypertext browsing does not scale well. We developed a query interface that serves as an extension of the browsing model of hypertext systems. These queries minimize the repeated user interactions required to locate data in a standard hypertext system. HyperFile is a prototype data server interface. In this article, we describe HyperFile, including a number of issues such as query generation, query processing, and indexing.

**Key Words.** Hypertext, indexing, user interface.

## 1. Introduction

Hypertext (Conklin, 1987) is emerging as a model for the management of loosely-structured information. The key idea is to view data as a collection of "cards" or nodes that are linked in a variety of ways. Each node may contain text or multimedia information. End users can view one or more cards at a time, and can traverse links to view other nodes.

Hypertext systems are currently built on top of file-based storage systems. This means that they often do not provide adequate data management facilities such as indexing, concurrency control, and recovery. Storage systems for hypermedia must provide these facilities (Halasz, 1988; Lange, 1992; Grønbaek, 1994). To add data

Chris Clifton, Ph.D., is Assistant Professor, Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208-3118, *clifton@eecs.nwu.edu*, Hector Garcia-Molina, Ph.D., is Professor, Department of Computer Science, Stanford University, Stanford, CA 94305-2140, *hector@cs.stanford.edu*, and David Bloom, B.S., is with Anderson Consulting, 1345 Avenue of the Americas, Room 928, New York, NY 10105.

management facilities to a hypertext system, one can hard code the facilities into an existing system, or have the hypertext system store its data in an existing database management system. We feel the second approach is far superior, since one does not have to "reinvent the wheel" in every hypertext system built.

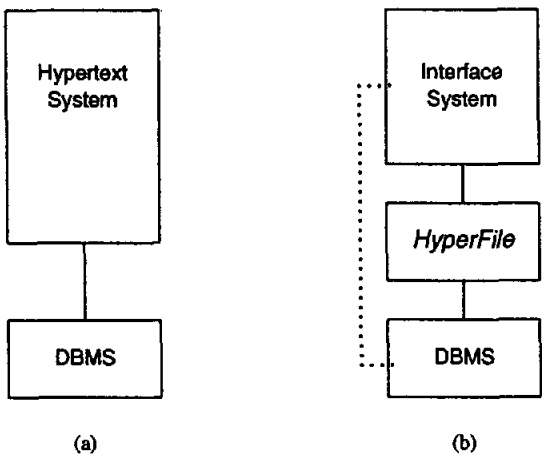
To run a hypertext system on top of an existing database management system, there are two options (Figure 1). Under option (a), a monolithic hypertext system interacts directly with the DBMS. The hypertext system must map its objects into the data model supported by the DBMS, either relational or object-oriented. To manipulate data, say to retrieve a particular node, the hypertext system must generate the appropriate query to the underlying DBMS. Option (b) differs from option (a) in that the hypertext system has been split into two components. The Interface System handles all interactions with the end user, rendering nodes on the screen, presenting menus to the user, showing "buttons" for traversing a link, and so on. What we have called HyperFile is a system that captures the common data management functionality needed by most hypertext interfaces. The key idea is that a single generic HyperFile system can tailor the data services offered by the even more generic DBMS to better serve a particular class of applications, in this case, hypertext applications. (Note that we do not rule out direct access to the DBMS by the Interface System. This is shown by a dotted line in Figure 1(b).) The term "blade" has recently been coined (Stonebraker, 1993; Ubell, 1994) to refer to this type of add-on system that enhances a DBMS for a particular class of applications. (This is analogous to how a blade is added to a razor.[1]) Using this notion, HyperFile can be viewed as a blade for hypertext management.

There are at least four types of functionality that HyperFile can provide:

1. *Data Model.* HyperFile presents to the Interface System a hypertext data model. From the point of view of the underlying DBMS, this "model" is simply a class library or pre-defined schema.
2. *A Query Language.* The common queries that the Interface System must generate are captured more easily and naturally in this language than in the underlying DBMS query language. HyperFile translates the incoming queries into DBMS queries, in much the same way as C++ is translated into C by some compilers.
3. *Indexing Facilities.* HyperFile can implement indexes that speed up the commonly expected hypertext queries. These indexes enhance, rather than replace, the index facilities provided by the underlying DBMS. These HyperFile index structures are stored in the DBMS, which treats them as application data.
4. *Distribution Facilities.* If the hypertext objects are stored on a collection of independent DBMSs, HyperFile can provide transparent access to these objects. If the underlying DBMSs are already integrated into a distributed

1. The implication is also that a company will make more money selling the blades than selling the razor. Our discussion here is independent of the financial aspects.

**Figure 1. Monolithic hypertext system vs. HyperFile**




DBMS, the distribution facilities may be unnecessary. However, in many cases, the underlying DBMSs are not integrated at the database system level (perhaps because they are from different vendors), so HyperFile can provide the necessary "glue" without requiring changes to the DBMSs. This glue can come in the form of a naming framework for distributed hypertext objects, facilities for following remote pointers, and facilities for indexing distributed collections of objects.

Typically, the Interface System (Figure 1(b)) runs on an end-user workstation, while the DBMS runs on one or more shared back-end servers. Although HyperFile could conceptually run at either the front or back end, we believe it is advantageous to run it at the back-end server where the DBMS resides. This is because a single HyperFile query may generate multiple DBMS interactions, examining much more data than the query ultimately returns. By placing HyperFile at the back-end, we can achieve an additional significant improvement over the monolithic approach of Figure 1(a): The data intensive hypertext search operations are performed tightly coupled to the DBMS, as opposed to transferring large volumes of data over the network for processing at the front-end.

The key to the success of a system like HyperFile is that it captures the data model and services needed to simplify the design of the Interface System. In addition, its indexing facilities should improve performance of common queries. In this article, we present the design of HyperFile and argue that it does satisfy the above criteria. We have implemented a prototype version of HyperFile and an Interface System, and have shown that HyperFile does significantly simplify the Interface system. We have also evaluated the performance of HyperFile's indexing facilities, and identify the cases where substantial improvements occur. We have also implemented and evaluated distribution services for HyperFile, but these will not

be discussed here (they were described by Clifton and Garcia-Molina, 1991). Final validation of HyperFile and the blade concept in general will, of course, come only over the years. However, we feel that this article presents an important first step: it gives a detailed description of a hypertext blade and its functionality. We believe that this article can serve not only hypertext applications, but other application areas as well, by showing in detail one case study of a blade and the design and performance issues involved.

We compare HyperFile with other approaches to managing Hypermedia databases in Section 3; we first give a description of the data model and query language we provide. Following Section 3 we discuss certain key aspects of HyperFile in detail:

- A Query Processing algorithm for HyperFile queries is given in Section 4.
- Indexing of HyperFile queries is discussed in Section 5.
- User Interface ideas and experiences are given in Section 6.

The Eiffel object-oriented language was used as an implementation vehicle for the prototype HyperFile server. This has given us considerable flexibility in modifying the prototype as we have developed new ideas, and it also shows the viability of implementing HyperFile on an Object-Oriented DBMS. This prototype runs on a variety of platforms, and has been used for experiments with various aspects of HyperFile. In Section 5.5.1 we discuss results of experiments with indexing. Section 6 also makes use of this prototype in conjunction with a sample application.

## 2. Data Model and Query Language

What is the right data model for a hypertext application? There is a spectrum of choices. At one end, we could have a very rich model, with many object classes. For instance, we could have one object type for textual nodes, another for image nodes, another for table of contents nodes, another for hypertext link nodes (giving information about the link), and so on. For each object type, the model would define the desired fields. For instance, a text node could have a date field, a body field, an author field, and so on.

However, by preordaining the types of objects and their structure, a rich model makes it hard to deal with diversity. For example, some Interface Systems may require additional or fewer fields within an object of a given type. Document structure is somewhat free-form; users will often find that the predefined structures do not fit their needs (this problem occurs in areas other than Hypermedia, see Zdonik, 1993).

We propose, instead, an object model at the other end of the spectrum. There is a single object class, but this class is "freeform." Essentially, each object is simply modeled as a set of tuples of the form <*tuple_type, key, data*>. Each tuple represents a property of the node, with the *tuple_type* being its name, the *data* being its value, and *key* being a short property that distinguishes that value from others of the same

**Figure 2. HyperFile data model**

```
{ (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 > ) }
```

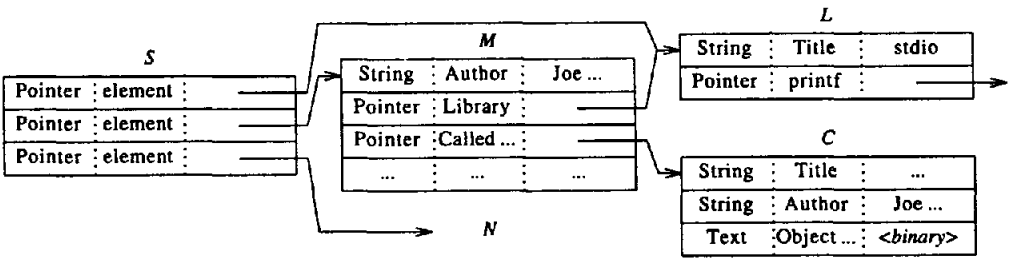
tuple type. These tuples can contain text, pictorial data, keywords, bibliographic information, references and pointers to other objects, or arbitrary bit strings. A sample set containing, for example, a module from a Software Engineering system, is given in Figure 2.

The system keeps a *Tuple Type Definition Table* that lists the allowable tuple types, giving for each the permitted data types that can occur in the key and data fields. For example, the tuple type `String` shown in Figure 2 must be defined in the Tuple Type Definition Table; its entry would specify that a tuple of this type must have a string key and a string data value. The `Pointer` tuple type is defined to have a key of data type "string" and a data component of type "object ID." An application can add entries into the Tuple Type Definition Table. For example, an application could define `Object_Code` to be a tuple type where the key would name the target machine (a string), and the data (binary) would be the actual object code. This would be a convention between applications; HyperFile would only understand `Object_Code` as a tuple type having a string as a key, and arbitrary bits as data. The data model does not understand (or restrict) the concepts of "target machine" or "object code" (except that the basic representation of "target machine" is a string, and of "object code" is a sequence of bits). Tuple type definitions extend across the HyperFile database, which encourages the sharing of data between interfaces. In a sense, the Tuple Type Definition Table represents the "schema" of the application databases.

Tuples may contain pointers to other objects, as shown in the above example. It is also possible for an application to use multiple HyperFile objects and pointers 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 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.

We can also use this capability to create sets of documents (used in set filter queries; Section 2.1.2). 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 3 shows a set *S* containing three objects: *M* (from the previous example), *N,* and

**Figure 3. Set of routines from a software engineering application**




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 $C$. This representation has a number of advantages over having a special *container* type for sets:

- The query language has a single set of operators. Every object in the system is of the same class.
- Sets can be permanent, in the same manner as any object is made permanent. This allows users to build "private libraries."
- It is easy to build annotated bibliographies. Since a set is an ordinary object, associating text, keywords, and other information with it is easy (just add descriptive tuples to the set).
- A paper that contains references can also be used as a *set of the referenced documents.* This allows easy "literature search" operations.

The set operations provided for individual objects also have the appropriate meaning for sets of objects defined in the above fashion. Since two sets $S$ and $T$ are actually sets of triples, where each triple points to an object in the set, $S \cup T$ produces a new set of triples which points to all of the objects in either $S$ or $T$. In fact, the primary use of these operations is likely to be on documents which are considered to be "sets of objects" rather than on individual items.

In most cases, a HyperFile database will contain a root set of all the objects in the database, much like a library card catalog. This allows searches over the entire database. However, the use of sets allows the scope of queries to be restricted if desired. This has a number of uses: A single query could construct a set on which a variety of further queries can operate; a user can repeatedly restrict the items of interest without having to repeat queries; or a query could operate on an already existing "mini-database." The root set could also serve as the root of a directory (the objects in root would be sets), allowing a hierarchical structure of the data.

As discussed at the beginning of this section, the HyperFile data model we are proposing is extremely simple. The disadvantage is that Interface Systems that

require richer object types will have to build them out of simpler objects. The advantage is that with a simple model, HyperFile is able to serve a wider class of hypertext applications. Furthermore, with the appropriate query language (such as the HyperFile Interface Language described later in this section), dealing with "collections of simple objects" should be as simple as dealing with richer objects.

To illustrate these points, consider the "pointer" triples illustrated in Figure 2. The pointer has a key that defines its meaning, for instance, whether it points to a "bibliographic reference" or a "called routine." These keys can be used for searching.

However, the Interface System cannot conveniently attach more information to a pointer (e.g., the date it was created and the name of the creator). To do this, the Interface System can create an auxiliary object. This object contains the relevant properties for the pointer (e.g., creation date, creator's name). The pointer in the original document points to the auxiliary object, and it in turn points to the referenced object. This also makes it possible to have a back pointer to the original document. As we will see in Section 2.1, our query language easily allows searches over both types of pointers (e.g., either following pointers with key "bibliography," or following pointers that in their auxiliary object have a certain creation date). In summary, Interface Systems that require a richer structure than what is provided by the basic model can provide it; Interfaces that do not need this richer structure are still able to user HyperFile.

Note that objects are represented as sets, so triples are not ordered within an object. This restriction substantially simplifies our language. Ordering can be obtained by linking the components together (e.g., part A points to part B which points to part C). As an alternative, ordering can be indicated by using a number in the key field.

A brief summary of the HyperFile data model is:

Object ← {Triple}
Triple ← (Tuple_Type, Key, Data)
Tuple_Type ← *identifier*
Key ← *Date* | *Numeric* | *String* | *Pointer*
Data ← *Date* | *Numeric* | *String* | *Pointer* | *Binary Large OBject*

Identifiers that appear as tuple types must exist in the Tuple Type Definition Table, whose entries have the following structure:

Tuple_Type_Entry ←<*identifier*, Key_Data_Type, Data_Type>
Key_Data_Type ← "string" | "numeric" | "date" | "pointer"
Data_Type ← "string" | "numeric" | "date" | "pointer" | "BLOB"

### 2.1 HyperFile Query Language

The Hyperfile Interface Language (HIL) is used to represent queries. By implementing the HyperFile data model as a user-defined type on an extensible database

system (Stonebraker and Rowe, 1986), we can also define new operations on that type, allowing a query language that reflects the needs of the users.

Much of the motivation for HyperFile queries comes from the browsing techniques of hypertext (Conklin, 1987). Browsing provides a very unrestricted method for searching; users of hypertext databases will often find things in ways not foreseen by the database designer. The problem with browsing is that it is labor-intensive; selection is done by manually navigating through the data. This is apparent with the World-Wide Web (Berners-Lee et al., 1992).

We use ideas from the information retrieval world to solve this problem. Rather than browsing through a large set of items, users issue queries that *filter* this set and produce new sets (much the same idea as the computerized card catalogs in some libraries). These techniques can limit the user to what the database designer believes to be useful queries; however we go beyond traditional Information Retrieval systems in that links in the data can be used by the queries to produce new sets "on the fly." Utilizing the links within the queries allows many of the benefits of browsing without the time consumed by the step-by-step "manual" approach. These queries allow the user to construct a small set of potentially interesting objects, which can then be viewed using a browsing approach. Manual browsing will also be helpful to give the user ideas as to how to define a query; however HyperFile queries can eliminate considerable manual interaction in between these two browsing phases.

Our filter queries provide for the common queries we expect to see in hypertext applications. As a matter of fact, we interviewed a number of potential users of a hypertext interface to determine what constraints their requirements would place on the storage system for a Hypermedia system. These users included hardware designers, programmers, hypertext users, and users of other document retrieval systems (Clifton et al., 1988). From our discussions we learned that chained queries (combining separate filtering criteria into a single query), pointer dereferencing 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.

The most interesting class of HyperFile queries are filter queries. The types of retrievals performed by these queries fall into two categories:

- Retrieval along pointer chains. This is important both for references and for retrieving parts of objects. These queries are the major difference between *hypertext* and conventional databases.
- Searches for objects meeting particular criteria. These are related to conventional database queries. The queries will look for specifics like keywords. They may also look for types of relationships between items (particular patterns of pointers to other objects).

In addition to filter queries (which retrieve multiple objects), we need queries that can manipulate individual objects. These Basic Filters are queries that retrieve selected triples from within an item. For example, we may desire the *abstract* (a tuple within a document) rather than the entire document.

**Figure 4. Syntax of top-level of HyperFile query**

query ::= expr → *object* *Basic Query, produces a new object*

expr ::= *object*
::= (*tuple_type, key_value, data_value*) *The set consisting of the given literal tuple*
::= expr setop expr *Basic set operations: Union, Intersect, etc.*
::= expr basefilter *Basic filter, described in Section 2.1.1*
::= expr setfilter *Set filter, described in Section 2.1.2*

setop ::= ∪,∩, -

The Basic set operations (union, intersection, difference) are also provided. These take an object (or set of objects), and return a new object (or set) without modifying the original. Changes are made to a single object with these functions as well.

It must be remembered that the HIL is embedded in a host programming language. Object identifiers are actually stored in variables in the host language. The HIL is not in itself a "complete" programming language, nor is it intended as a user interface. It is a *query* language for use by programmers writing a Hypertext interface system.

The rest of this section describes specific features of the query language. Although most of the features are covered, this is by no means a user manual for the language. We will briefly mention some basic operations, then go into detail on filtering queries. Finally we will describe how data is transferred between HyperFile and applications. We will also give a running BNF for the HIL; the top level (including basic set operations) is shown in Figure 4.

*2.1.1 Basic Filters.* These are operations that take an object away, and return a new object that includes a subset of triples of the original. They are based on triple selection using pattern matching. Perhaps it is easiest to start with an example. Given a document (object ID) *D*, we can construct a new object consisting of just the authors of the original document as follows:

```
D(string, "author", ?) → object ID
```

This is the *triple selection filter.* Note the use of the `?`, a pattern matching character that matches any data item. It can also be used in the key or tuple type fields. Standard range expressions are also allowed for basic data types (date, numeric, regular expressions for strings).

Filters can also be joined using `and`, `or` and `not`. For example,

**Figure 5. Syntax of a HyperFile basic filter**

| | | |
|---|---|---|
| basicfilter | ::= basicfilter basefilter | |
| basefilter | ::= (typespec, key, data) | |
| | ::= basefilter *and* basefilter | |
| | ::= basefilter *or* basefilter | |
| | ::= *not* basefilter | |
| key | ::= matching-expr | |
| data | ::= matching-expr | |
| matching-expr | ::= *literal of appropriate type* | |
| | ::= *expression of appropriate type* | |
| | ::= *?* | |
| | ::= application-communication | *Described in Section 2.1.3* |
| typespec | ::= *name of type of this triple* | |
| | ::= application-communication | *Described in Section 2.1.3* |

```
D( (string, "author", "Chris*") OR (string, "author", "Hector*") )
    → object ID
```

returns author triples in *D,* which have either Chris or Hector as the prefix of the data. Figure 5 contains a description of the syntax of basic filters.

*2.1.2 Set Filters.* Set filtering queries are used when the user has a large set of *potentially* interesting objects (perhaps the entire database), and wishes to find a small set of items which are *actually* of interest. We assume that the user (or application) has some idea of what makes an object interesting, and how the user would manually browse the database given the time (which links would be followed, etc.). The first of these gives the criteria on which to filter (much as in an Information Retrieval system). The second gives the *scope* of the query, and allows HyperFile to follow links in the manner of hypertext browsing.

In particular, 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 (for 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 3), 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*); then 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.* Processing is analogous to Unix pipes: the objects in *S* flow through a series of filters (in this case a single one), and the objects that satisfy the conditions in the filters end up in *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. In the next part of the query, the values placed in each *X* are dereferenced using the operator $\uparrow\uparrow X$.[2] This adds *C* to the set of "possible results" (which becomes $\{M, N, L\} \cup \{C\}$.) 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 an item, 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.

More complex comparisons are allowed. For example, we may wish to find articles with multiple authors:

---

2. 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.

```
S | (String, "Author", ?X) | (String, "Author", X ≠?Y) → T
```

If an object has only a single Author tuple, `X` will be set to the name in the data field of that tuple. The second part of the filter will also select the same tuple and bind `?Y` to the data field. Since `X=Y`, the tuple does not match and, as there are no other author tuples, the object does not pass this filter. In the case of a document with two author tuples (with names *Chris* and *Hector*) the first part of the query will bind both names to *X*. The second part of the filter will test a tuple (say the one with author *Chris*) and find that there is a binding for `X` (*Hector*) which is not *Chris*, and the tuple will match. Since at least one tuple matches, the object passes the filter and is placed in the result set `T`.

The occurrence of the variable preceded by ? specifies that it is free; without the ? it is bound. Filters are evaluated left to right, hence the leftmost occurrence of a variable should be a free occurrence (otherwise nothing will match). Further free occurrences add to the set of possible values for the variable for that object. Figure 6 contains a description of set filters. Another way of thinking of matching variables is that each instance of an object passing through a filter has its own set of variables. Each variable is actually a set of values, which it matches in that object. An expression using the variable is true if any of the values in the set would make the expression true. A more formal understanding of matching variables can be obtained from the query processing algorithm in Section 4.

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 a primary 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. With HyperFile, a query can be developed following the browsing style of the user: *If I were to browse, I would follow links to called routines, looking for those authored by "Joe Programmer."* This is then performed with a single request to the server. The HyperFile query language is designed around queries that simulate browsing; operations are provided to mimic browsing while allowing considerable server flexibility in the way the operations are processed. We believe that queries like the above are representative of those occurring in a Hypertext environment and, hence, must be handled efficiently and naturally.

**Figure 6. Syntax of Hyperfile Filter Queries**

```
expr          ::= ...
                  ::= expr setfilter

setfilter     ::= [ setfilter ]^n
                  ::= [ setfilter ]*
                  ::= setfilter setfilter
                  ::= | filter

filter        ::= selector
                  ::= filter or filter
                  ::= not filter
                  ::= arrow

selector      ::= (typespec, key, data)
                  ::= selector arrow
                  ::= selector selector

arrow         ::= ↑ filtervar
                  ::= ↑↑ filtervar
                  ::= filtervar

key           ::= matching-expr
data          ::= matching-expr

matching-expr ::= literal of appropriate type
                  ::= expression of appropriate type
                  ::= expression involving matching variable
                  ::= application-communication

matching-variable ::= ?
                  ::= ?filtervar

filtervar     ::= identifier
```

The query language we have defined so far does not permit "joins" between sets. For example, say we have two sets of documents, $S$ and $T$, and we wish to identify documents that have a common author. This cannot be expressed with a single HyperFile query. An application program would have to extract the authors in $S$ (see Section 2.1.3) and then search for those authors in $T$. Although our query language could be extended to include joins, we have not done so because we believe such queries are rare in hypertext applications. The added complexity to

the query language and the query processing algorithms outweighs the benefits of supporting rarely used queries.

*2.1.3 Transferring Data to the Application.* The preceding queries do not illustrate how results are actually provided to the application. Providing just an object identifier to the HyperFile server will return the entire object (a basic filter with no selection criteria). It is also possible to retrieve certain fields as part of another query (for example, gather the titles of all items found in the query).

To do this, values of fields in a tuple are retrieved explicitly using the $\rightarrow$ 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, a C application program could contain:

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

to display individually all of the titles of documents (neatly numbered) in `S` written by `Chris Clifton`.

The above variable `title` can be of any data type in the applications programming language. HyperFile sees this data only as a string of bits (although the type may be limited based on the tuple_type field, as described on page 5). 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.[3] 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.

The translation from a 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.[4]

---

3. The interface between a large HyperFile field and the host language is dependent on the host language. As an example, the prototype (which uses Eiffel, an object-oriented language, as the host language) returns an object of a type that inherits from FILE, and can be treated as a file by the applications program. We could return a socket (file pointer) to a C language application.

4. 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 it 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.

**Figure 7. Syntax of communication with applications**

| | |
|---|---|
| application-communication | ::= → *host_variable* |
| Used in matching_exper; | *place value in host language variable.* |

Note that the above retrieval is explicit; filtering queries that only search for objects of interest will return the result set, but not any of the data in the objects in that set. This gives the number of items in the result; the user can then decide if further queries are needed to restrict this set. These queries need not send large amounts of data (e.g., text, bitmaps). When the set of items of interest is small enough that the user actually wants to see the items, a query is issued to retrieve just the desired fields. (The syntax is described in Figure 7.)

We have not discussed queries that 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 that 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.

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. 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 in which to look: in the current module, in all called modules, or in the entire program containing the current module.

## 3. Comparison With Other Systems

In this section, we briefly compare some common data storage systems to a storage system composed of HyperFile and its underlying DBMS (Figure 1(b)). We will argue that, for loosely-structured hypertext information, HyperFile provides better tailored services.

### 3.1 File Systems

HyperFile is probably most similar to a file system, particularly one with *self-describing data records* (Wiederhold, 1987). 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 desirable. 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. File systems are inefficient for search and retrieval. In a large (and particularly 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 (Data Base Task Group, 1974) in that they both provide objects and pointers. 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, as discussed in Section 5), 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 that 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

Information retrieval systems provide powerful means for accessing text (Salton, 1989). The main difference between HyperFile and information retrieval systems is the support of pointers. The ability to incorporate pointers as part of the "search space" is needed in a Hypertext database (Frisse and Cousins, 1989). Also, information retrieval systems typically do not support non-text data.

Ideas from information retrieval systems, combined with hypertext methods, can be used to form a general interface to a HyperFile database. Information retrieval research into automatic indexing (Salton, 1988), automatic structure detection (Salton et al., 1994), and natural language (Croft and Lewis, 1987) can be used to generate properties for textual objects and a flat document into a "semi-structured" HyperFile document.

### 3.4 Relational Systems

Relational systems provide a regular structure for data. HyperFile supports data that do not fit into a regular structure. Although work has been done on placing text items in a relational database (Stonebraker et al., 1983; Smith and Zdonik, 1986), creating a relational database that 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 (e.g., pointers, flexible data types), but we address these below.

### 3.5 Advanced Database Systems

Advanced database systems such as object oriented (Maier et al., 1986; Woelk et al., 1986; Weinreb et al., 1988) and extended relational (Stonebraker, 1986; Schwarz et al., 1986; Dadam et al., 1986) 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). As discussed in the Introduction, HyperFile is likely to be a "value added system" on top of an advanced object oriented storage system. It provides a data model, query language, and index structures that are specifically tailored to the needs of hypertext applications.

There are other examples of "blades" to support documents in database systems. Atlas (Sacks-Davis et al., to appear) builds a document model on top of a relational database. This system adds full-text search and nesting to support structuring to the standard relational model. Atlas provides the ability to store and query large collections of documents (including full-text search), and provides support for pointers. Atlas is perhaps best viewed as a "blade" for supporting information retrieval applications. In Christophides et al. (1994), a Standard Generalized Markup Language (SGML) data model is built on top of $O_2$ (Deux, 1991). The goal here is to represent a structured document in a database.

HyperFile is similar to these systems in providing support for collections of documents. However, the query mechanism is designed around *hypertext.* The support for path queries in HyperFile is much greater (in particular, supporting limited depth transitive queries and "pruned" paths) than in either of these systems. These location dependent queries allow the user to maintain the mental model of "browsing hypertext" while issuing queries.

### 3.6 HyperBases

Hypertext specific database management systems (HyperBases) are currently being developed. These systems need query facilities (Lange et al., 1992; Lange, 1993; Schnase et al., 1993; Wiil and Leggett, 1992). HyperFile is a proposal for a schema/query facility for these systems. We feel that HyperBase management systems should be built using existing DBMS technology, HyperFile attempts to show that this is feasible.

### 3.7 $G^+$

$G^+$ is a graph query language developed at the University of Toronto (Cruz et al., 1987). It has goals in common with HyperFile, and provides a more powerful query language. Like HyperFile, $G^+$ provides for graph based transitive-closure queries. However, computing some $G^+$ queries can be NP-hard (Mendelzon and Wood, 1989). This defeats our goal of providing a simple and efficient back-end data storage service. By concentrating on browsing-style object retrieval queries we can keep our language simple (an advantage for both computational and interface development reasons). We hold that support for sophisticated analysis of the relationships between data items (as opposed to retrieval based on those relationships) is not required by users of hypertext systems.

## 4. Query Processing

Basic filters and other basic operations are straightforward to process. The algorithm for processing filtering queries is more interesting. It is worth noting that the design of the query language has allowed a simple and efficient processing algorithm for filtering queries, as described in this section.

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 (possibly the result of an expression), $S_\theta$ is the result set of objects, and each $F_i$ is a filter operation (setfilter) 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 $F_i$, 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 that contain the keyword *Indexing*. In addition, follow reference pointers for

three levels searching for objects that meet these criteria.

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

In the above query, $F_1$ = (*pointer, Reference, ?X*), is a selection operation that 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, Indexing,* ?) does simple pattern matching: Any object containing a tuple with type *keyword,* key *Indexing,* 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 $S_\theta$.

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

The basic means for processing queries is to create a *working set W* containing objects in the original set *S.*[5] 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

5. 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 Kapidakis (1990) shows that a node-based search (such as a breadth-first search) will give the best results in the average case.

match, the object passes the filter. The pattern can be a variety of things; "Matching" depends on the pattern:

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, such as in the example in Section 2.1.2. 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 pattern matching is straightforward but dependent on the data type of the field being compared.

```
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 ($\uparrow$ 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

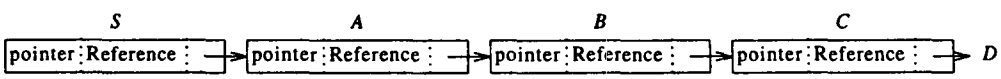
Some of the initialization of $P$ in the above needs explanation. *P.next* is set to the filter after the dereference. *P.mvars* starts empty; the set contains no bindings. The use of *P.start* and *P.iter#* will be explained in the next paragraph.

- 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.[6]

6. $O.iter\# \geq k$ is not tested if $k = *$. $*$ may be thought of as $\infty$.

**Figure 8. Chain of references**




$E(I_j^k, O)$ :

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 are 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 working 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 8). We will run the following query (described at the beginning of this section) on the set $S$:

```
S [ | (pointer, "Reference", ?X) | ↑↑ X ]^3 | (keyword,
    "Indexing", ?) → 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 *Indexing* and adds $A$ to $T$ if the keyword is found. $B$ is then removed from the set, and processing starts 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*$\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 that cover the transitive closure of a graph of pointers (queries that 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 9 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 that must be maintained in $W$ is 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*.

This also adapts well to distributed query processing. This is done by mapping the object ID into the location of the object. When an object is dereferenced, it is tested to see if it is local. If not, instead of adding it to the working set $W$, it is sent to the site that contains the item (along with the query $Q$, and the initial site that started processing $Q$). When a site receives such a message, it is either already processing the query (in which case it just adds the new item to its working set), or it must create a working set and begin processing the query. Results from the query are sent directly back to the site which originated the query (which may not be the site that sent the reference to be processed). This is only a brief overview; for a complete description of the distribution issues in HyperFile see Clifton and Garcia-Molina (1991).

**Figure 9. Query processing algorithm**

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

## 5. Indexing

As with many large databases, some HyperFile queries can take considerable time to process. A query which searches every item in the database can take time that an interactive user would consider unreasonable. Indexing speeds up these searches by effectively "precomputing" parts of common queries. Indexing HyperFile queries is somewhat different from standard indexing techniques. This is because the scope of a query is determined by the pointers in the data, rather than being statically determined by the database schema. Extensible indexes (Stonebraker, 1986; Aoki, 1991) allow user-defined indexing and access methods in advanced database systems.

For example, a traditional relational index returns tuples in the indexed relation based on a search key. Any query which is based on that key with a scope of the entire relation can make use of the index (actually, in some cases the index could be useful even if the scope were a part of the relation, such as a view which involves a selection from that relation). The scope of the query can be determined simply from looking at the query and the schema (database catalog). With HyperFile, the scope of a query may be dependent on the *contents of the objects.* A standard index over the entire database may return hundreds of objects for a given search key; determining which of these objects are in the scope of the query may be more expensive than performing the query without the index. However, the HyperFile query processor can select a HyperFile specific index that is appropriate to the query.

Our indexing technique starts with the simple idea of attaching an index to an object in the database. The index allows lookup of items based on a particular

attribute type (the *property* of the query), and covers objects which could be reached from that node following a particular type of link in a "browsing" interface (the *scope* of the query).

### 5.1 What is Indexed

The choice of a key for indexing can be quite varied; just about any type of data will serve. This is no different from indexing in a traditional database. However, specifying the *scope* of the index is different. Rather than specifying a relation or set which is to be indexed, we must specify a portion of the graph: a place from which queries will start, and a type of link to follow. Creating an index thus requires specifying three parameters: The *anchor point* (node) to which the index is to be connected, the *search key* for the index, and the *link type* which determines the scope of the index.

Figure 10 is a sample database consisting of two types of links (solid and dashed) and a single attribute (noted as *key*). An index has been created at node *root* on the attribute *key* and the link type solid. A few interesting points to note about the index are:

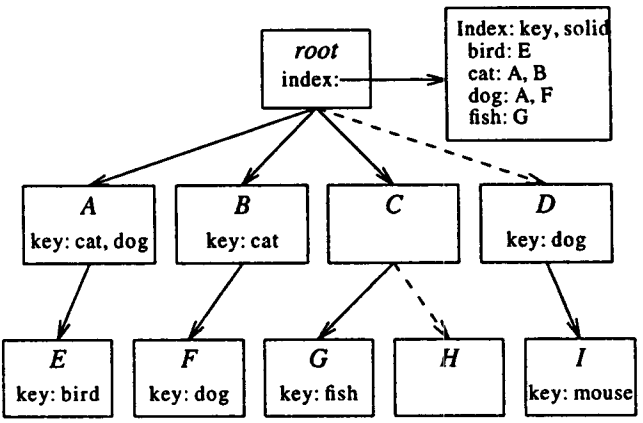
- Item *D* is not in the index, even though it has a key of interest. This is because the index is for items reachable through solid links, and *D* is reached only through a dashed link.
- Item *I* is pointed to by a solid link. However, since it is not *reachable* from *root* via solid links, it is not in the index.
- Item *G* is in the index, even though its parent (*C*) does not appear in the index. Node *C* is in the scope of the index, but does not appear since it has no *key* attribute.

The index of Figure 10 will speed up searches whose scope is the solid-link tree rooted at *root*.

### 5.2 Structure of the index

The index itself is structured in a manner similar to a traditional database index. *B*-trees, hashing, and other such techniques are all applicable. However, certain special information is required. In addition to pointers from the index to relevant objects, objects are required to have back pointers to indexes which *potentially* include them. This is necessary to maintain the index properly. For example, in Figure 10, *C* will have a back-pointer to ensure that updates that add keys to *C* will be reflected in the index. Items *D, H,* and *I* do not need back pointers, as changes to these objects will not result in their being reachable, and thus they will not be in the index. If the dashed links are changed to solid, the presence of back pointers to the index in the parents of the links will point to the need for including *D, H,* and *I* in the index.

**Figure 10. Index of a tree-structured database**



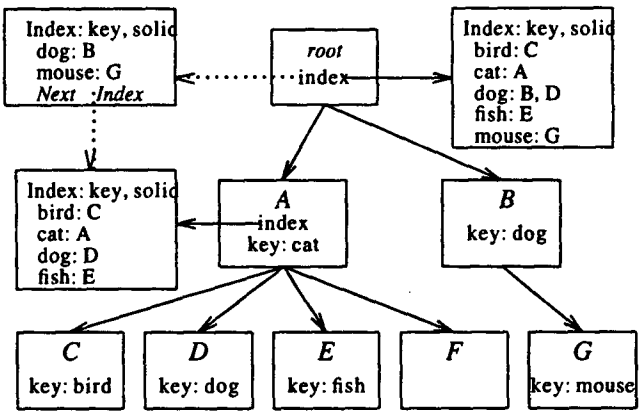

### 5.3 Multiple Indexes

In a real system, there may be many nodes from which we often make queries. We could build an index at each of these nodes, but this leads to space problems due to replication of information. Figure 11 provides an example of this situation. Some users may wish to query the entire database, using index *root*; others may only be interested in the subset contained in the tree rooted at $A$. To allow the efficiency provided by indexing to both sets of users, we can construct indexes anchored at both nodes (the indexes pointed to by *solid* lines.) All of the functions described at the end of the previous section will work here as well. Note that each object which is below $A$ must have back-pointers to both indexes.

This naive approach has one problem. All of the items in index $A$ are also indexed by root. This leads to replication in the indexes. In a large database with many indexes, the size of the indexes could in fact grow at a faster rate than the size of the database itself. Given that the index grows linearly in the number of items indexed, a complete set of indexes on an $n$ node tree of depth $k$ would take space $O(n \cdot k)$. A more space-efficient index structure would help, but the indexes could still end up requiring more space than the data itself. In addition, updates to the database may take a long time because they must modify many indexes.

This replication can be eliminated by requiring indexes to refer to "lower" indexes, rather than directly indexing the entire subtree (illustrated by the indexes pointed to by *dotted* lines on the left side of Figure 11). A search for all items in the database (starting at *root*) that have attribute *dog* would first find $B$ from the *root* index. Next, the search would proceed along the *Next Index* pointer to the index anchored at $A$, where it would find $D$. Note that this increases the time required to

**Figure 11. Tree-structured database with two indexes**




find an item. In the worst case, putting an index at every node, we end up with a linear search and have lost the benefits of indexing. However, we expect the typical cost will be much smaller.

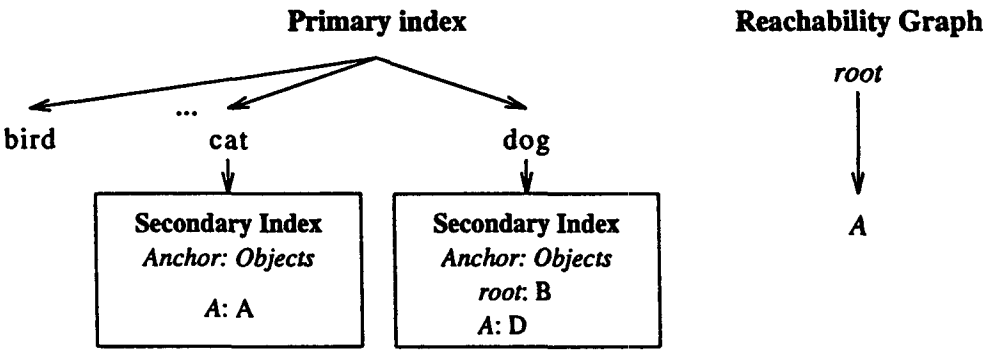
Update in such a system is slightly more complex, although the time required is less (due to updating only a single index). This complexity results from the need to remove links between indexes when links between objects are changed, in much the same manner as objects must be removed from the index in the basic scenario.

### 5.4 Single Multiple-Attribute Index

An alternative to the previous structure is to use a single database-wide index for each type of key. In a sense, this is a multiple attribute index (Lum, 1970). However, the second attribute in our system is "reachability," rather than an attribute in the normal sense. As such, previous techniques do not apply.

Our method is to use a single *primary* index on the search key that returns a *secondary* index. The secondary index maps the "anchor points"(nodes in the database which have indexes) to the objects that can be found from those anchor points. The structure of the primary and secondary indexes could be any of a number of things, including $B$-trees, hash tables, and sorted lists. A naive implementation of the secondary indexes, in which each anchor point hashes to a list of all of the objects reachable from that anchor point, could require $O(n^2)$ space per secondary index (where $n$ is the size of the database). However, all of the objects at many anchor points are reachable from other anchors (e.g., in Figure 11 all objects reachable from $A$ are also reachable from root). This fact was used to eliminate replication in the previous section. In the secondary index, we can associate with a given anchor

**Figure 12. Single multiple-attribute index**




point only those objects for which it is the "closest" anchor point, cutting the space considerably (worst case $O(n)$). A number of algorithms which can be used for our secondary index are presented in (Jagadish, 1990).

For example, Figure 12 is a sample index containing entries for a few keywords based on the database of Figure 11 (with anchor points at *root* and $A$). Note that the secondary index for "dog" only associates $B$ with the anchor *root*, even though a query on "dog" from *root* would also find $D$. Node $D$ is associated with the anchor point $A$. The reachability graph on the anchor points is used to determine which anchors can be reached from the desired "start" anchor point. The result set of data items is then the union of all of the nodes found from all of these anchors (in the chosen secondary index.) To illustrate a search, say that we wish to find all of the objects reachable from *root* which contain the keyword "dog." We use the primary index to find the secondary index associated with "dog." We also need all of the anchor points reachable from *root* (done using the reachability graph, these are *root* and $A$.) Next, we find all of the objects reachable from these anchor points using the secondary index. The objects $B$ and $D$ are the result of our search.

### 5.5 Cost Comparison

The methods of indexing we have introduced (single indexes, indexes with replication, indexes without replication, and multiple-attribute indexes) each have advantages and disadvantages. We have computed simple estimates of the time and space costs for each technique on a regularly-structured database. Some comparisons of the indexing techniques based on this analysis are presented here.

First, we set out the assumptions we used for these estimates. Although the techniques work for an arbitrary directed-graph structured database, we assume that the data is tree-structured. The structure of data in a hypermedia database is likely to be oriented towards a tree more than, for example, a randomly-created directed graph. We feel that worst-case costs derived for tree-structured data will reflect practical costs better than an analysis on arbitrary graph-structured data. Another assumption is that searches will only use indexes at or below the start node.

For the purposes of this discussion, we assume that the data and pointers are indexed form a complete tree with constant branching factor (each parent has the same number of children). This restriction significantly simplifies the analysis, and we feel the analysis on this structure will reflect performance on more varied data. The Tektronix HyperModel Benchmark (Anderson et al., 1989) uses such an arrangement as one of its three "hierarchies." Later in this section, we present experiments on less regularly structured data, and compare the results with the results of the analysis.

We will use indexes placed at the root and at all nodes halfway down the tree. This provides a uniform placement of indexes (each index has an equal number of nodes located "directly" beneath it). Such an arrangement is an intuitively reasonable example. We will also look at a single index placed at *root*, as described in Section 5.1.

We assume a logarithmic index structure (such as a $B$-tree), and a linear time to search the data otherwise. Parameters to the analysis include the total number of possible keys $K$, the probability $P$ that a given key attribute value appears in a given data item, the branching factor $B$ (number of children of any non-leaf data item), and database size $N$.

We have worked out formulas which give an estimate of the time and space requirements of the indexing methods described (Clifton and Garcia-Molina, 1990). Here we present graphs which show estimates based on those formulas.

The graphs in this section are based on complete trees with a branching factor of five. We did try varying the branching factor; the results varied by an equivalent factor for all of the indexing methods. The values of $K$ and $P$ are given above each graph. We assume a main memory database; with increasing (Gigabyte) memory sizes we expect to be able to cache some information, such as links and keywords, for each node in the database. Access to disk will only be needed to obtain complete objects, which will not be required for queries which search large portions of the database.

For each of the indexing methods, Figure 13 shows the find time for a search over the entire database. We use $K$=1,000 and $P$=.001, which provides an expected value of 10 search keys per node.

Figure 14 shows the expected time for queries from just below the root of the database (thus encompassing one fifth of the database). Otherwise, this figure corresponds exactly to Figure 13. The gains provided by indexing are substantial.

**Figure 13. Find-time vs. database size, search from root**

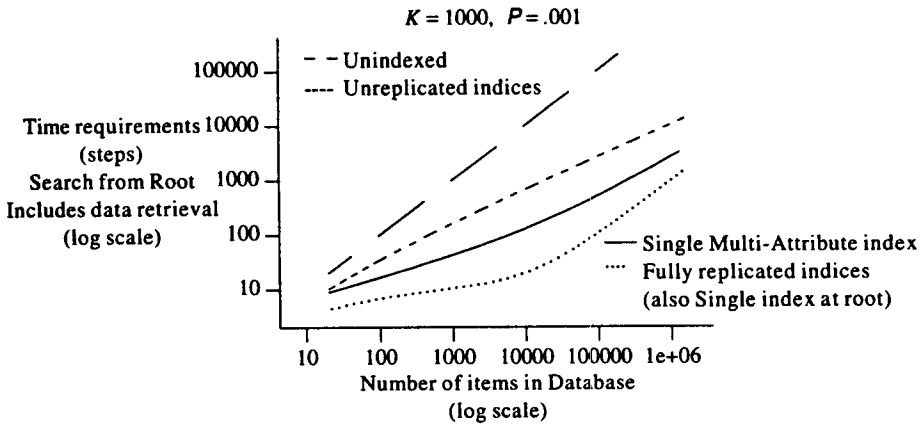
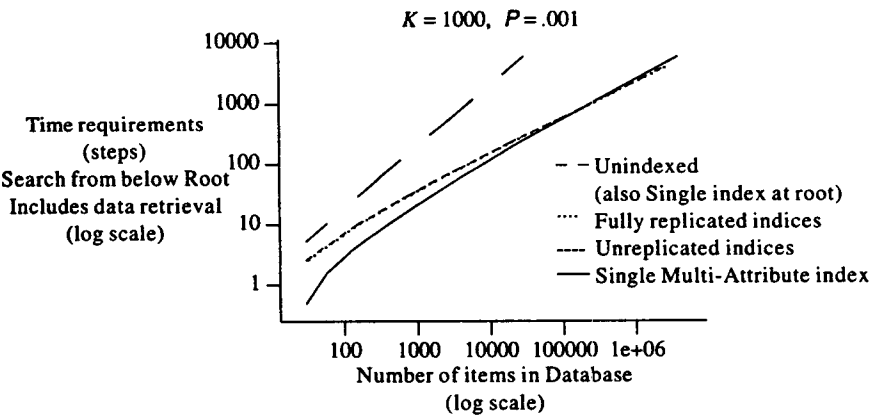



**Figure 14. Find-time vs. database size, just below root**




*5.5.1 Experimental Results.* The preceding discussion of costs assumes a very regular database. Practical databases will have a more varied structure. We believe that the cost functions of the previous section will be reasonably close to costs on practical databases. We have performed some experiments using our prototype query processor/main memory database on less regularly structured databases to verify this. We include graphs in this section, which plot the experimental results alongside predicted results from our analytical estimates.

The experiments presented here serve two purposes:

- To verify our analysis.
- Perhaps more interestingly, to explore how well we can predict indexing performance on data which do not hold to the strict structure of the analysis (complete trees with a fixed branching factor).

The experiments presented here were run on a DEC 5410 with 128MB of main memory. This allowed us to run experiments with many nodes, while still caching all of the query information in main memory (to provide a meaningful comparison with the main memory analysis).

To perform these experiments, we must first calibrate the model (determine the actual index lookup and database search time). We assumed that the time to search through the database (without an index) was linear in the size of the database; based on this we determined that a search took $3ms$ per item. The index used for our experiments is a balanced binary search tree. We determined that the time to look up an item in an index of size $n$ is $\log_2(n) \cdot 1.5ms$.

To see how well our analysis predicts performance on databases without a regular structure, we performed experiments on randomly constructed databases. Note that the databases used in the experiments are not entirely random collections of nodes and links. We expect large hypertext databases to have a structure which resembles a tree more than, for example, a completely connected graph. Therefore, our experiments are based on data with a somewhat regular structure. The databases were built within the bounds of the following parameters:

- Each node contains a single key, randomly selected from a space of 700 distinct keys.
- The number of outgoing branches from each node varies randomly from 1 to 7.
- Each path from the root to a leaf node is at least of length four.
- For the tests on indexed databases, each database has an index at *root,* and indexes at each node "halfway" between the root and the leaves (using the fully replicated index method described in Section 5.3).

This forms a tree-structured database. Although not presented here, queries on this database were quite close to the predictions from our analysis. Of more interest is what happens when we relax the requirement that the database be a tree. To do this, we added new links to the databases described above, which formed a directed acyclic graph. Specifically, from each node $N$ in the database, we added a number of links to children of the siblings of $N$. Note that this corresponds to the *PartOf* relationship of the Tektronix HyperModel benchmark (Anderson et al., 1989). The number of outgoing links from each node was selected randomly from 1 to 7. We assigned a different *link type* to these new links; the experiments on these databases used only links of the new type.

The following graphs contains data points for identical sets of queries run with and without indexing. Each data point corresponds to a different database, and represents an average time of forty queries on that database. Note that each point represents an average of queries on a single database rather than an average over several databases of the same size; we are interested in seeing the deviation in a particular database from the prediction of the analysis. The lines represent the theoretical results from the analysis of the previous section, with a branching factor

**Figure 15. Queries from root, DAG database**

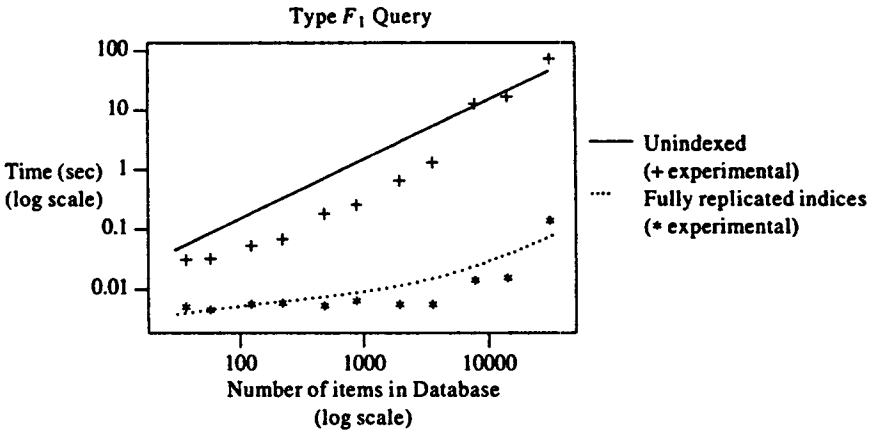
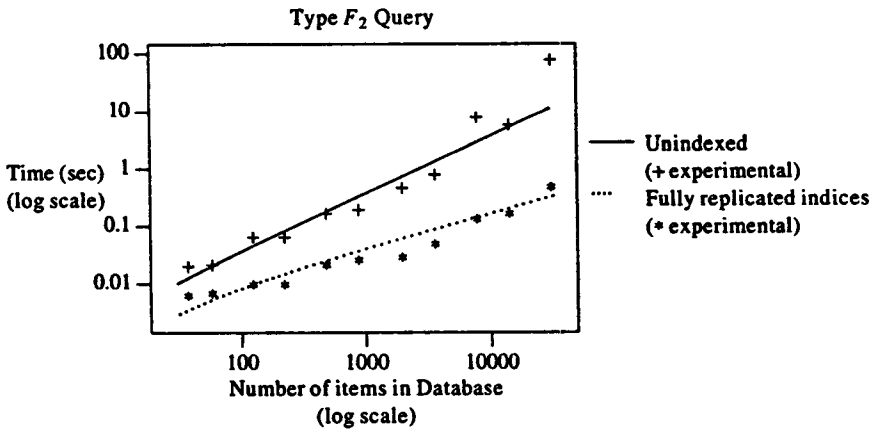



**Figure 16. Queries from just below root, DAG database**

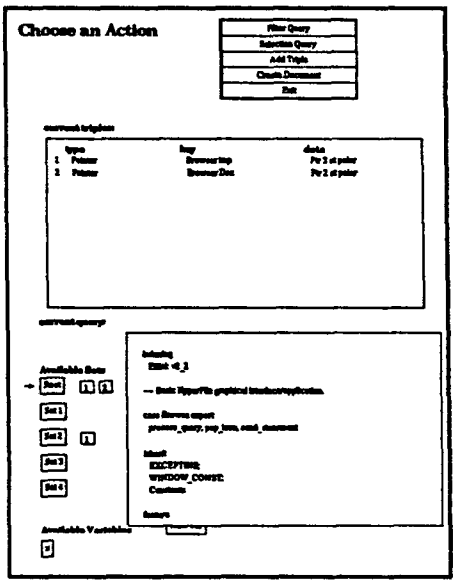



$B$=4 (the parameters on key placement are $K$=700 and $P$=1/$K$, which correspond exactly to the experimental databases).

Figures 15 and 16 show the results of queries run over these databases. Although not exact, the predictions do appear to be in the ballpark, particularly with the larger databases. Of more importance, the prediction of performance improvement appears quite close; if the experimental index is slower than predicted, so are the experimental results without an index.

The trends in the experiments coincide relatively well with the predictions from the analysis. The model we described earlier in this section cannot be used to predict the exact performance of indexes on a particular database. However, the model can be used to study tradeoffs and general trends.

**Figure 17. Complete browser screen**




## 6. A Sample Query Generation Interface

It is expected that HyperFile user interfaces will be application specific. For example, the kind of interface desired for a CAD/CAM database would be combined with a CAD design tool, while an on-line library application would likely resemble a hypertext browser. Different applications will result in different kinds of queries, and this will change the way in which the user interface is used to generate queries.

We have built an interface for gaining experience with HyperFile query generation and use. The interface presented here is not intended as THE application for HyperFile. It is instead an example of ideas that might be incorporated into more application-specific interfaces. This application was written using the Eiffel object-oriented programming language and runs under the X window system.

The interface we have developed runs in a single application window. Figure 17 contains a sample screen. (All figures showing the application window are actual screen dumps). Conceptually, we have divided this window into three horizontal regions. The top region of the window contains an area for menus, as well as a "prompt message." The center region is used for display of results; in a production system, this would be application specific, for example it could be a traditional hypertext browser. The lower region of the screen contains a number of sets (`Root, Set1,...`); these are used to store the results of queries (and as starting sets for further queries). To the right of some of these sets are small boxes; these represent the items in the set. Clicking on the set "button" will display the contents

of the set in the center region; clicking on one of the small boxes will retrieve and display the contents of that particular item. To the right of these is a text output window; this appears on demand to display long, unstructured (text) fields. This would be subsumed by application-specific means of output in production systems. At the bottom are matching variables, and variables used to retrieve fields during a query.

To demonstrate how the browser works, we use a database that contains this section of this article, as well as the implementation of the browser. Note that this article is linked to the implementation and vice-versa, thereby allowing a user reading about the screen layout, for example, to look at the code defining this layout. We first build the following query to recursively find routines called by the main program of the browser (to two levels):

```
Root [ | (Pointer, 'Calls', ?X) | ↑↑ X ]2 → Z
```

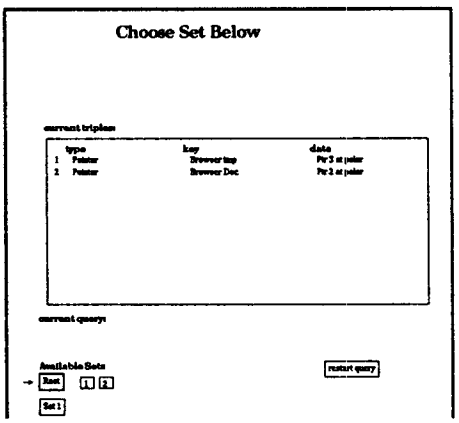
We then look through these routines for those written by David, and take a look at the code of one of those routines with the following query:

```
Z | (String, 'Author', "David Bloom") | (Sources, 'Eiffel', → code)
    { display_text (code) }
```

Instead of typing queries, the browser lets us enter queries interactively, using menus. The menu at the top of Figure 17 offers a number of options:

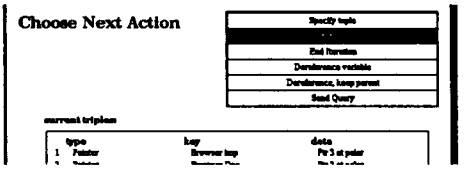
- Filter Query: Search the database for objects meeting specified criteria.
- Selection Query: Choose specific tuples from an object.
- Add Triple: Add a tuple to an object.
- Create Document: Create a new (empty) object.
- Exit.

We select `Filter Query` and are then prompted for a set of items to start with. For our first sample query, we start with the `Root` set (which contains the top level of the Browser program, as well as this article):

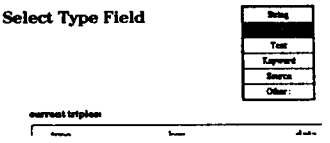


Selecting `Root` requires a simple mouse click on the button marked `Root` (at the bottom of the preceding illustration).

We now choose the criteria on which we wish to select:




In this case, we will start by iterating (since we will want to follow pointers recursively). We are then returned to the same menu, and choose `Specify Tuple`. This allows us to specify criteria, which restrict the objects we are interested in; objects that do not meet these criteria will be ignored. In this case, we want to follow `Pointers` to `Called` routines. We are immediately prompted for the *tuple_type* of the tuple on which we wish to search:




Note that the tuple_type menu is application specific; it could be hard-coded into the browser, or perhaps "gleaned" from the database catalog.

Following this, we are given options for the *key*.

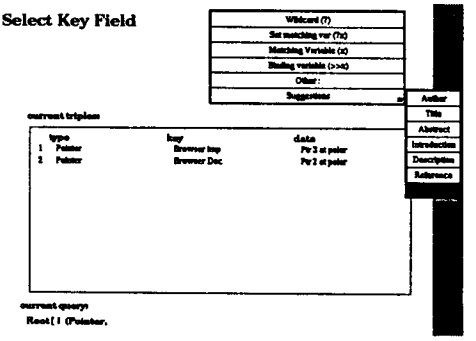
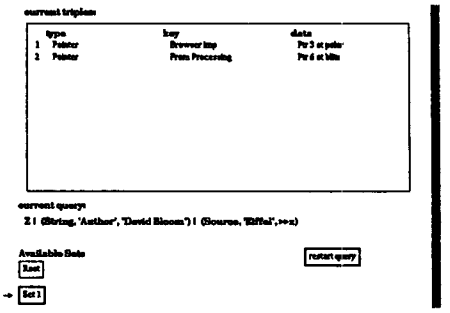


**Figure 18. Result of recursive query**



Note that we have a number of options:

| | |
|---|---|
| Wildcard: | Accept any key. |
| Set Matching Var: | Set a variable which can be used later for comparisons (such as ?X in the query language). |
| Matching Variable: | This tests the value of a matching variable. |
| Binding Variable: | This sets a variable which can be later viewed (but not used in a query). |
| Other: | This is a chance to enter your own value, if none of the given options are appropriate. |
| Suggestions: | This is a submenu of application-defined "interesting possibilities." |

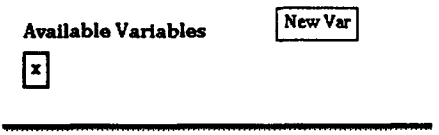
In this case, we just pick `Calls` from the suggestions menu.

It is also worth noting that the partially completed query is displayed as it is constructed; this is shown at the bottom of the preceding figure (under `current query:`).

We next have to specify the data field. In most cases, this is a long field, such as the text of an article. As a result, comparisons will be infrequent. However, in the case of a *Pointer* tuple, the data field contains the pointer to another object. Since we want to dereference this pointer (for a tuple with key `Calls`), we will set a matching variable:

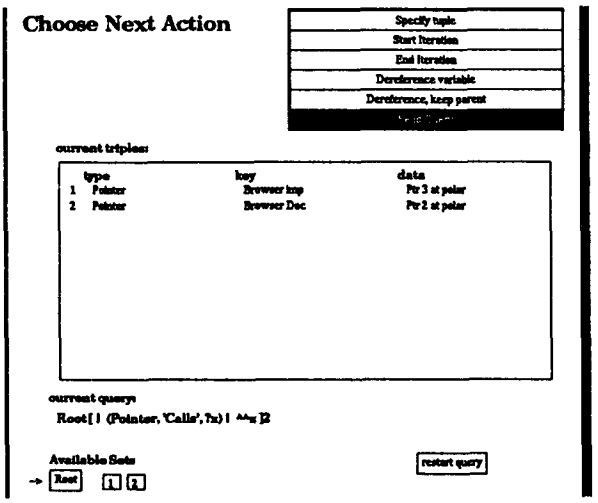

An unused identifier is assigned for this variable, and inserted in the `Available Variables` area (shown at the bottom of Figure 18).

Now we have completely specified the tuple for this filter. This returns us to the `Next Action` menu. We have picked out the pointers we want to follow, so we choose `Dereference, keep parent`. This adds all of the pointers in a matching variable to the objects being processed. Of course, we have to specify the matching variable from the list at the bottom of the window:




Note that `x` was added to this list when we set the matching variable using the data menu above.

All that remains is to specify that the loop (recursively chasing pointers) is done; to do this we choose `end iteration` from the next action menu. This prompts for the number of iterations (keyboard entry): An integer (for a fixed number of iterations) or * for a complete transitive closure. We will only go two levels deep (there is no sense in gathering the entire code just for an example). We are ready then to send the finished query:




The results of this query are displayed in the window at the top of Figure 18. The result contains two tuples, each of which is a pointer to another object (note the contents of the `data` fields). The results are also placed in the next available set (in this case `Set1`) for future use. Currently "next available" is the least recently used set; other options (such as allowing the user to specify which set) could be used. An arrow points to `Set1`, to show that it is the currently displayed set (as shown in the bottom of Figure 18).

## Figure 19. Viewing a binding variable

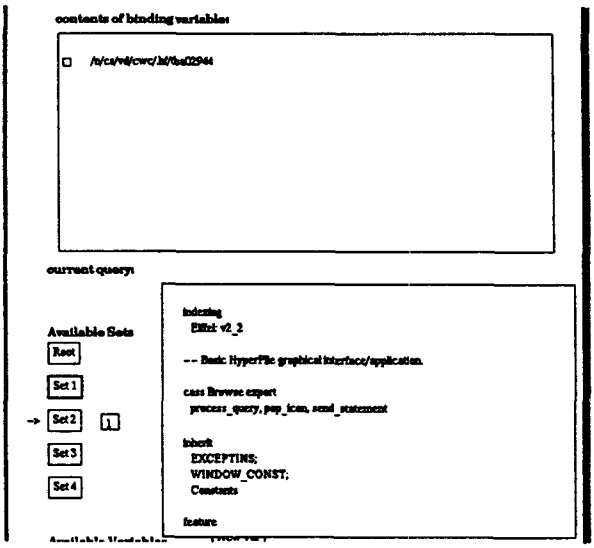



Figure 18 also shows the next query. As previously mentioned, this is to find all of the programs in the result of the previous query (placed in `Set1`, referred to by the letter `Z` in the display of the query) which were written by David, and view the source code from those programs. This is constructed in the same manner as the previous query. A few interesting differences:

- The name of the author was not selected from a menu, but was entered by choosing `Other:` from the data menu (which prompts for keyboard input). Likewise for the language type (the key of the `Source` tuple).
- The data from the source tuple are retrieved explicitly ( (*Source, 'Eiffel',* $\rightarrow$ *X*) ), to allow later viewing.

Once the query has been executed, we can view the contents of the binding variable used to explicitly retrieve the program text. To do this, we simply click on the variable (at the bottom of the window). This brings up the contents in the main viewing area (in this case, a single value as shown in Figure 19). Also notice the small box to the left of this value; this means that the item is actually a text field, and clicking on this box allows us to view it in a separate window (in this case it is the text of a program).

Note that the results were placed in `Set2`, as the arrow is currently pointing to it. The set is not displayed in its entirety in this figure, as we are looking at the binding variable. However, one of the tuples in `Set2` is a pointer; we can see this because of the small box to the right of `Set2`.

The user interface we have presented allows the construction of arbitrary HyperFile queries. An actual application would probably not be as general: instead

providing "canned" query parts which would be combined by the user to form the actual query. These query parts would be given in application-specific language rather than displaying the actual HyperFile query. We believe the idea of menu-based query construction and hints serves as a good basis for forming application-specific queries.

## 7. Conclusions and Further Work

We have described HyperFile, a data model and query facility 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 hypertext applications can be answered with a single request to HyperFile. Yet, HyperFile is simple and flexible enough that designers of such applications will not have to resort to file systems to store their data to avoid frequent schema changes (and corresponding modifications to the applications).

This article presented the HyperFile query processing algorithm. We have also looked at developing indexes to support HyperFile queries. A browsing style of user interface has also been presented (more as an idea of what can be done than as a specific application).

We are currently looking at further applications of HyperFile, such as scientific databases. Scientific environments are often heterogeneous in hardware, software, and perhaps most importantly, personnel. Although such environments have made use of traditional business-oriented databases, rarely are all of the data put into such a database. Scientists within an organization will often create special-purpose systems for their own use. Although HyperFile will not eliminate such systems, it provides the capability to store and link data, code, and notes so that the information will still be available to future researchers.

## Acknowledgments

Much of the motivation for this work comes from the needs of hypertext researchers at Xerox PARC (Halasz, 1987; Halasz, 1988). Some of the ideas described in this article were initially developed at Xerox in discussions with Robert Hagmann, Jack Kent, and Derek Oppen. We would like to acknowledge their contribution.

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 the first author was supported in part by an IBM Graduate Fellowship.

## References

Anderson, T.L., Berre, A.J., Mallison, M., Porter, H., and Schneider, B. The Tektronix HyperModel benchmark specification. Technical Report No. 89-05, Tektronix Computer Research Laboratory, Beaverton, OR, August 3, 1989.

Aoki, P.M. Implementation of extended indexes in POSTGRES. *SIGIR Forum,* 25(1):2-9, 1991.

Berners-Lee, T.J., Cailliau, R., Groff, J.-F., and Pollermann, B. World-wide web: The information universe. *Electronic Networking: Research, Applications, and Policy,* 2(1):52-58, 1992.

Christophides, V., Abiteboul, S., Cluet, S., and Scholl, M. From structured documents to novel query facilities. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Minneapolis, MN, 1994.

Clifton, C., Garcia-Molina, H., and Hagmann, R. The design of a document database. *Proceedings of the ACM Conference on Document Processing Systems,* Santa Fe, NM, 1988.

Clifton, C. and Garcia-Molina, H. Indexing in a Hypertext Database. *Proceedings of the International Conference on Very Large Databases,* Brisbane, Australia, 1990.

Clifton, C. and Garcia-Molina, H. Distributed processing of filtering queries in HyperFile. *Proceedings of the IEEE International Conference on Distributed Computing Systems,* Arlington, TX, 1991.

Conklin, J. Hypertext: An introduction and survey. *IEEE Computer,* 20(9):17-41, 1987.

Croft, W.B. and Lewis, D.D. An approach to natural language processing for document retrieval. *Proceedings of the Tenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval,* New Orleans, LA, 1987.

Cruz, I.F., Mendelzon, A.O., and Wood, P.T. A graphical query language supporting recursion. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* San Francisco, CA, 1987.

Dadam, P., Kuespert, K., Andersen, F., Blanken, H., Erbe, R., Guenauer, J., Lum, V., Pistor, P., and Walch, G. A DBMS prototype to support extended $NF^2$ relations: An integrated view on flat tables and hierarchies. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Washington, DC, 1986.

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

Deux, O. The $O_2$ system. *Communications of the ACM,* 34(10):34-48, 1991.

Frisse, M.E. and Cousins, S.B. Information retrieval from Hypertext: Update on the dynamic medical handbook project. *ACM Hypertext Proceedings,* Pittsburgh, PA, 1989.

Grønbaek, K., Hem, J.A., Madsen, O.L., and Sloth, L. Cooperative hypermedia systems: A dexter-based architecture. *Communications of the ACM,* 37(2):64-74, 1994.

Halasz, F.G., Moran, T.P., and Trigg, R.H. NoteCards in a nutshell. *Proceedings of the ACM CHI+GI Conference,* Toronto, Canada, 1987.

Halasz, F. Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *Communications of the ACM,* 31(7):836-852, 1988.

Jagadish, H.V. A compression technique to materialize transitive closure. *Transactions on Database Systems,* 15(4):558-598, 1990.

Kapidakis, S. Average-case analysis of graph-searching algorithms. Ph.D. Thesis, Princeton University, Princeton, NJ, 1990.

Lange, D.B., Østerbye, K., and Schütt, H. Hypermedia storage. Technical Report R-92-2009, The University of Aalborg, Institute for Electronic Systems, 1992.

Lange, D.B. Object-oriented hypermodeling of Hypertext supported information systems. *Proceedings of the Twenty-sixth IEEE International Conference on System Sciences,* Hawaii, 1993.

Lum, V.Y. Multiple-attribute retrieval with combined indexes. *Communications of the ACM,* 13(11):660-665, 1970.

Maier, D., Stein, J., Otis, A., and Purdy, A. Development of an object-oriented DBMS. *Proceedings of the ACM Object-Oriented Programming Systems, Langauges, and Applications Conference,* Portland, OR, 1986.

Mendelzon, A.O. and Wood, P.T. Finding regular simple paths in graph databases. *Proceedings of the Fifteenth International Conference on Very Large Data Bases,* Amsterdam, 1989.

Sacks-Davis, R., Kent, A., Ramamohanarao, K., Thom, J., and Zobel, J. Atlas: A nested relational database system for text applications. *IEEE Knowledge and Data Engineering,* to appear.

Salton, G. Automatic text indexing using complex identifiers. *Proceedings of the ACM Conference on Document Processing Systems,* Santa Fe, NM, 1988.

Salton, G. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer.* Reading, MA: Addison-Wesley, 1989.

Salton, G., Allan, J., and Buckley, C. Automatic structuring and retrieval of large text files. *Communications of the ACM,* 37(2):97-108, 1994.

Schnase, J.L., Leggett, J.J., Hicks, D.L., Nuernberg, P.J., and Sanchez, J.A. Design and implementation of the HB1 hyperbase management system. *Electronic Publishing: Origination, Dissemination, and Design,* 6(1):35-63, 1993.

Schwarz, P., Chang, W., Freytag, J., Lohman, G., McPherson, J., Mohan, C., and Pirahesh, H. Extensibility in the Starburst database system. *Proceedings of the International Workshop on Object Oriented Database Systems,* Pacific Grove, CA 1986.

Smith, K.E. and Zdonik, S.B. Intermedia: A case study of the differences between relational and object-oriented database systems. *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications,* Orlando, FL, 1986.

Stonebraker, M. Inclusion of new types in relational database systems. *Proceedings of the Fourth IEEE International Conference on Data Engineering,* Washington, DC, 1986.

Stonebraker, M. The Miro DBMS. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Washington, DC, 1993.

Stonebraker, M., Stettner, A., Lynn, N., Kalash, J., and Guttman, N. Document processing in a relational database system. *Transactions on Office Information Systems,* 1(2):143-158, 1983.

Stonebraker, M. and Rowe, L. The design of POSTGRES. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Washington, DC, 1986.

Ubell, M. The Montage extensible DataBlade$^{TM}$ architecture. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Minneapolis, MN, 1994.

Weinreb, D., Feinberg, N., Gerson, D., and Lamb, C. An object-oriented database system to support an integrated programming environment. *IEEE Data Engineering,* 11(2), 1988.

Wiederhold, G. *File Organization for Database Design.* New York, NY: McGraw-Hill, 1987.

Wiil, U.K. and Leggett, J.J. Hyperform: An extensible hyperbase management system. Department of Computer Science Technical Report No. TAMU-HRL 92-003, Texas A&M University, College Station, TX, 1992.

Woelk, D., Kim, W., and Luther, W. An object-oriented approach to multimedia databases. *Proceedings of the ACM SIGMOD International Conference on the Management of Data,* Washington, DC, 1986.

Zdonik, S.B. Incremental database systems: Databases from the ground up. *Proceedings of the ACM SIGMOD International Conference on the Management of Data,* Washington, DC, 1993.