HyperFile, a Database Manager for Documents

Christopher Wade Clifton

A Dissertation
Presented to the Faculty
Of Princeton University
In Candidacy for the Degree
Of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

June 1991

Abstract

Documents, pictures, and other such non-quantitative information pose interesting new problems in the database world. Such data has traditionally been stored in file systems, which do not provide the security, integrity, or query features of database management systems. We have developed HyperFile, a data server that provides query facilities (as well as some other database features) while maintaining the flexibility and efficiency of a file system.

HyperFile is based on the *hypertext* notion of free-form objects connected by links. Hypertext systems "query" their database by *browsing* (reading objects and following links.) We present a query interface that maintains much of the flavor of browsing, allowing the user to specify a single query rather than manually following links. This eliminates the repeated user interactions of hypertext browsing, and allows the hypertext model to be extended to larger and less structured databases.

An algorithm for processing HyperFile queries is presented. We also show how to extend this algorithm for distributed query processing, and present experimental results from a distributed HyperFile server.

Another issue explored is indexing. In HyperFile, searches are often demarcated by pointers between items. Thus the scope of the search may change dynamically, whereas traditional indexes cover a statically defined region such as a relation. This demands new indexing techniques. Some ideas on indexing in HyperFile are presented, as well as experiments in a large HyperFile database.

Also presented is a sample HyperFile application. This is a "browser" that uses menus to guide the user in constructing HyperFile queries.

Keywords: Database, Hypertext, Indexing, Query processing.

Acknowledgements

I owe thanks to many who have provided help and inspiration with this work. I will not try to list everyone, however a few stand out:

Hector Garcia-Molina, as my advisor, has been involved in this work from the start. He has done more than simply oversee this work, however. He has taught me much about the responsibilities of a faculty member; I hope that I have learned enough to see me through the next phase of my career.

The database and distributed systems community at Princeton has provided a framework for my ideas; many design decisions have been influenced by their work. I will not attempt to name everyone, however Rafael Alonso had provided many suggestions both where my work relates to his and as a reader of this dissertation.

Andrew Appel also deserves credit for the comments he has made as a reader of this thesis. I hope I have incorporated all of his suggestions, as the thesis has been substantially improved by his comments.

Neal Young has allowed me to bounce ideas off of him and pick his brain when my work pushed the boundary of “systems” research (and my expertise), as well as providing technical help on numerous occasions. For this I thank him.

Finally my family, who have tried to keep me sane and convince me that someday this work would be done. In particular I wish to dedicate this thesis to my grandmother Evelyn Bonney, who strongly influenced my decision to pursue the course that has brought me to this point.

Table of Contents

CHAPTER 1

Introduction

HyperFile is a back-end data storage and retrieval facility for *document management* applications. The goal of HyperFile is not just to store traditional documents containing text. It also supports multimedia documents containing images, graphics, or audio. In addition, it must support *hypertext* applications where documents are viewed as directed graphs and end-users can navigate these graphs and display their nodes. [1] Another goal is to provide a *shared* repository for multiple and diverse applications. For example, it should be possible for a user running a particular document management system to view a VLSI design stored in HyperFile. Similarly, a user running a VLSI design tool should be able to refer to a document that describes the operation of a particular circuit.

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

[1] Much of the motivation for this work comes from the needs of hypertext researchers at Xerox P.A.R.C.[Hala87, Hala88]. Some of the ideas described in this thesis were initially developed at Xerox in discussions with Robert Hagmann, Jack Kent, and Derek Oppen. I would like to acknowledge their contribution.

interest, most of the searching can be done at the server, while at the same time being straightforward enough to have a simple and efficient implementation.

Given our requirements, it makes sense to implement HyperFile as a back-end service, as shown in Figure 1.1. Although not essential, we do expect that in many cases applications and HyperFile will run on separate computers. This is because:

1: HyperFile represents a shared resource so it is important to off load as much work as possible.

2: The applications probably have different hardware requirements (e.g., color graphics displays) than the service (e.g., large secondary storage capacity, high performance IO bus.)

3: Separate machines enhance the autonomy of the applications.

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

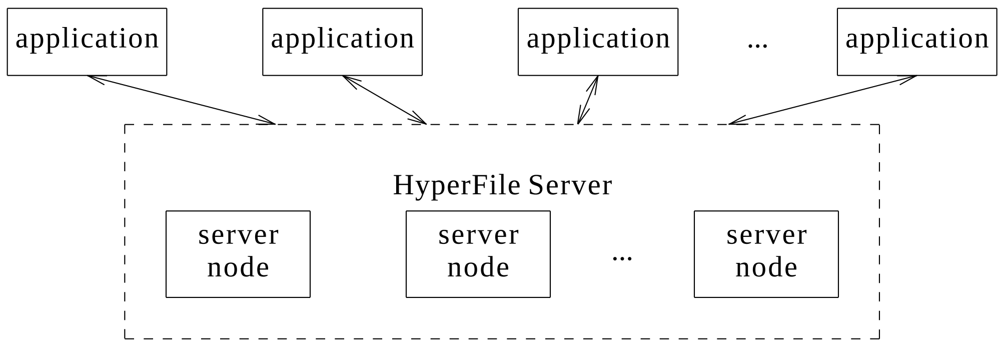


Figure 1.1: HyperFile *as a back-end service.*

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

At the other end of this server spectrum we have advanced databases, such as extended relational and object-oriented. These provide added functionality, but at the expense of increased complexity and rigidity. Advanced databases typically add structure (e.g. relations, object schemas, attribute inheritance) that makes it hard to manage irregular data or data that does not follow the (predefined) schema. This increased complexity also introduces a performance overhead (more complex algorithms, schema processing, advanced type checking.) Users who do not need the higher functionality must still pay these costs. HyperFile avoids these problems by providing loose structure and a limited set of features. This allows HyperFile to act as a high speed data server; added functionality can be provided by the applications.

In our server interface spectrum, there are of course other options in addition to files, advanced database systems, and HyperFile. We feel that they do not meet the goals we have for a data server. Other options are surveyed in Chapter 3, we will first give a more detailed overview of HyperFile in Chapter 2. Nevertheless, at this point we do want to stress that we are not ruling out other interfaces for different applications (or even for document processing ones). As a matter of fact, other interfaces (such as an object-oriented database or a file system) could be implemented at the server next to (or even on top of) HyperFile. Our point is that HyperFile represents an interesting point in the interface spectrum, providing the right mix of facilities and simplicity for many document management applications.

Following Chapter 3 we discuss key aspects of HyperFile in detail:

- A Query Processing algorithm for HyperFile queries is given in Chapter 4.
- Distributed HyperFile is discussed in Chapter 5. This includes extensions to the query processing algorithm, as well as the mechanics of keeping track of distributed data.
- Indexing of HyperFile queries is discussed in Chapter 6.

- Chapter 7 discusses certain other issues, such as version mechanisms, and ideas on how to best implement a “production” HyperFile system.
- User interface ideas and experiences are given in Chapter 8.

A prototype HyperFile server has been built. The Eiffel object-oriented language was used as an implementation vehicle; this has given us considerable flexibility in modifying the prototype as we have developed new ideas. This prototype runs on a variety of platforms and has been used for experiments with various aspects of HyperFile. Section 3 of Chapter 5 gives results of experiments with a distributed HyperFile server. In Section 7 of Chapter 6 we discuss results of experiments with indexing. Chapter 8 also makes use of this prototype in conjunction with a sample application.

We do not discuss all of the issues that would need to be addressed in building a production HyperFile; many problems such as crash recovery and data integrity are not substantially different from existing database systems and discussing the approach taken in HyperFile would introduce little that is novel. We instead concentrate on problems where HyperFile requires different solutions than existing systems.

Note: Some of the work in this dissertation has been previously published. In particular, Chapter 2 contains some information that appeared in[Clif88]; Chapters 4 and 5 include material from[Clif91]; and Chapter 6 contains work that was presented in[Clif90].

CHAPTER 2

Model and Interface Definition

The goal of HyperFile is to provide a shared repository for diverse types of data. In order to meet this goal, we must not constrain what may be placed in the database. This requirement leads to the data model of a file system; data as simply a stream of bits. We wish to provide more capabilities than a file system, however this requires some understanding of the data. We cannot hope to have the data server understand the underlying representation of all of the information we wish to put in a HyperFile database; new data representations may be invented well after HyperFile is created (for example, video compression techniques.) We instead allow objects that are a collection of *searching information*, which is understood by the server; along with unstructured “bit streams”.

One of the primary capabilities provided by HyperFile is a query facility. These queries are based on the *browsing* idea of hypertext systems. Browsing involves looking at an object, and following a link based on the contents of that object. This seems to be an appropriate technique for loosely-structured information. Browsing does not scale well, however. As a hypermedia database grows the paths to the desired information may become long, requiring repeated user interactions (and potentially following many “dead ends”) in order to find the desired data. Careful construction of the database to avoid this problem defeats our goal of providing a flexible data repository. Instead HyperFile provides queries (described in Section 2.4) that allow specifying what one would do if *browsing* the database; search for specific properties, follow certain types of pointers, etc. The query then retrieves the desired objects using a single interaction with the data server.

This Chapter gives the data model and query interface language for HyperFile. The interface described here is not for user-level interactions; it is instead the language that governs communications between applications and the data server. The actual user interface will be application dependent, although multiple applications (and thus multiple user interfaces) may be used to generate queries for a single HyperFile database. The goal of this interface is to be able to represent the queries and structure that should be handled by the data server (primarily locating and retrieving data) while limiting the complexity of query processing at the server. Complex processing of the data is the responsibility of the application, and should not be performed by the server.

1. Document Model

A HyperFile object consists of a set of *triples*. Each triple contains a *type*, a *key*, and a *data item*. The triple type serves two functions: it identifies the purpose of the triple and defines the actual types of the key and data fields. The key is a structured field used for searching. The data field may be used for searching in some cases, but may also contain unstructured data such as text or pictures. The following is a sample document:

```
{ (string, "title", "The Design of a Document Database")
  (string, "author", "Chris Clifton")
  (keyword, "hypertext", 35)
  (keyword, "database", 76)
  (keyword, "hypertext", 83)
  (pointer, "reference", <pointer to a document>)
  (integer, "pages", 15)
  (text, "Introduction", "Lots of text goes here...")
  (contents, <pointer to a subsection>, 1)
  (contents, <pointer to a subsection>, 2) }
```

The first two triples record the fact that this document has two properties of type string. These properties are named (for search purposes) "title" and "author." The triple type string defines that both the key and data fields are strings, with the key probably being a short string of fixed length. This definition of the triple type string is stored by the system in a *catalog* (discussed in Chapter 7, Section 4.) These definitions are application dependent (rather than predefined by HyperFile), although type definitions extend across the system to encourages sharing between applications. The types are in some sense the “schema” of a HyperFile database, but serve as suggestions rather than constraints and thus preserve the flexibility we desire. The primitive types that are provided for key and data items are also discussed Chapter 7.

To demonstrate, in the preceding sample document the triple type keyword specifies that the triple contains a keyword (short string) and its relative importance to the object (integer percentage.) Note that HyperFile treats the third field simply as an integer, and it is up to the application to interpret this as a percentage. These keywords probably appear somewhere in the text triple, but HyperFile is not aware of this. The application is responsible for maintaining consistency between the text and the keywords.

The data model we are proposing here is relatively simple. One reason is that our objective is a common model for different applications, a type of “common denominator”. This means that we cannot expect a HyperFile server to understand the semantics of each object property. A second reason is efficiency. If the HyperFile server is to quickly examine large numbers of objects, the properties used for searching must be simple and compact. A third reason is that the complexity of the query interface is proportional to the complexity of the model. Since we desire a simple language (to be described in the next section), we require a simple model. Nevertheless, in spite of the model simplicity, we believe that it is sufficiently powerful to support the types of queries that will be of most interest on HyperFile data.

To illustrate these points, consider the pointer triple illustrated in our sample document. The pointer has a simple label that can be used for searches, but contains no other structure. If the application does attach more information to links (as in some hypertext systems), it can define a complex link type consisting of a simple pointer (in the *key* field) and an unstructured data field that encodes the desired information. When the application wishes to examine a link, the data field can be retrieved and examined. With this approach, however, the system cannot search on these link properties. If this is desired, a second option is to make the link an object in itself. In this case, the original object contains a simple pointer to the link object. The link then contains the relevant properties (e.g., date, name, color, etc.) as well as one or more pointers to other objects. In summary, applications that require a richer structure than what is provided by the basic model can provide it for themselves.

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 points to part C). As an alternative, ordering can be indicated by the key or data field, as illustrated by the last two triples of our sample document.

2. HyperFile Interface Language

The HyperFile Interface Language (**HIL**) is used to represent queries. We have discussed the overall objectives of the system. The queries we wish to support fall into two primary types:

- Searches for objects meeting particular criteria. These are related to conventional database queries. The queries will look for specifics like document keywords. They may also look for types of relationships between objects (particular patterns of pointers to other documents.)
- 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.

In order to achieve these goals we combine ideas from two areas. We start with the idea (from information retrieval) of taking an initial set of objects, and restricting this set based on specific properties of the object (keywords, presence of a video track, etc.) to obtain a smaller set of objects. In the information retrieval model, the initial set is often the entire database (such as a library card catalog.) Although we do not rule out such sets, we believe that HyperFile databases will often have smaller initial sets such as the papers one is working on or all of the material belonging to a particular research project. We also incorporate the hypertext/browsing idea of following pointers; this is used to extend the set of interesting objects.

In addition to queries that retrieve entire objects, we need queries that retrieve selected triples from within an object. For example, we may desire *abstracts* rather than entire documents.

Most query operations take an object (or set of objects), and return a new object (or set) without modifying the original. Changes to an object are made with functions that operate on a single item.

It must be remembered that the HIL is a query interface, where the HIL queries are generated by an application program. The HIL is not in itself a “complete” programming language. It can be used as an embedded programming language, where the object identifiers are actually stored in variables in the host language. As an example of another approach, we have implemented a HIL parser that ties into the Eiffel language. This was used in developing the Browsing interface to be described in Chapter 8. With this approach, the application generates a string containing the HIL query. The parser then sends this as a query, and returns objects that contain the results and appropriate variable bindings. A short example of this will be given at the end of this Chapter. It must be remembered that the HIL is not intended as a user interface. It is a *query* interface between HyperFile and the application.

We will first discuss some of the basic operations of the language. These are not particularly novel, nevertheless they are included at this point for completeness and as background material for the more interesting types of queries. The language features used to form more interesting queries, those that provide queries that extend the idea of browsing a hypermedia database, will be described in Section 2.4.

2.1. Set operations

Since objects are structured as sets, the basic set operations of *union* ($\cup$) *intersection* ($\cap$) and *difference* ($-$) are provided. Each binary operator takes two objects, and returns a new object (set of triples) as appropriate for the operation.

For example, $A \cup B$ would return a new document consisting of all the triples contained in A and all the triples in B. Note that A and B are variables in the host language, not the actual documents; the statement

$$\mathrm{A} \cup \mathrm{B} \rightarrow \mathrm{B}$$

places the object identifier of a new object into B; this new object is the union of the objects (sets of triples) originally pointed to by A and B. The original objects are unchanged.

The query operators are divided into two types; those that operate on single triples, and those that manipulate sets of documents. Most of the queries are built around *filters*. These process existing documents, creating new ones based on certain selection criteria.

2.2. Basic filters

These are operations that take an object, and return a new object that includes a subset of the triples in the original. They operate by looking for particular triples, primarily based on the triple type and key, and adding these to the new object.

Filters are based on triple selection using pattern matching. Perhaps it is easiest to start with an example. Given a document *D* (where *D* is a variable in the host programming language containing an object identifier) we can construct a new object consisting of just the authors of the original document as follows:

D(string, "author", ?) $\rightarrow$ *document id*

This is the *triple selection* filter. Note the use of the **?**. This is a pattern matching character, which in fact matches any data item. It can also be used in the key or type fields.

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

D((string, "author", "Chris*") OR (string, "author", "Hector*")) $\rightarrow$ *document id*

returns author triples in *D* that have either Chris or Hector as the prefix of the data.

2.3. Basic operations

The filter operations only provide for selecting documents, not modifying (or even creating) them. In addition to queries, there is a simple functional interface to the system. The simplest of these functions is *create document*, which returns a document identifier. The actual operation is:

Create $\rightarrow$ *object identifier*

The result of this function may be used in any manner appropriate in the host language; assignment to a variable would be a common use. Note that deletion is handled via garbage collection; for a created object to become permanent (after a “session” with an application terminates) it must be pointed to by something. The exception to this is a “Root” object. In typical databases, the Root will be a set of various application (and user) specific *base sets*, so that each user and application will have their own view of the database. This is in some sense similar to a hierarchical file system.

Note that copying can be accomplished by a basic filter with no selection criteria:

Source $\rightarrow$ *Copy*

There are also operations that can be used to make changes to existing documents. These work at the triple level. The basic ones are *add* and *delete* triple.

add_triple(*document id*, *triple_type*, *key*, *data*)
delete_triple(*document id*, *triple_type*, *key*, *data*)

A *modify* operation for triples could be added; currently this is done using delete and add.

2.4. Set filters

We can now present the queries that serve as the meat of the HyperFile interface language. Set filters are queries used to select items meeting particular criteria. For example, we may wish to find all documents by a particular author, or search through all documents referenced by a given paper. These operate by selecting objects out of a *set*, rather than the entire database. In many 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. The use of sets, however, allows the scope of queries to be restricted if desired. For example, we may have a set of just the programs and documentation of a particular software project, allowing queries about just that project.

Sets are actually a type of object. This is done using triples containing pointers. A set is simply an object containing pointers to other objects. *S* in Figure 2.1 is an example of a set containing the items *A*, *B*, *C*, and *F*. This representation has a number of advantages over using a separate data type for sets:

- The language has a single set of operators. Every object in the system is built on the same model.
- Sets can be permanent, in the same manner that an object is permanent.
- It is easy to build annotated bibliographies; since a set is an object, associating text, keywords, and other information with it is simple.

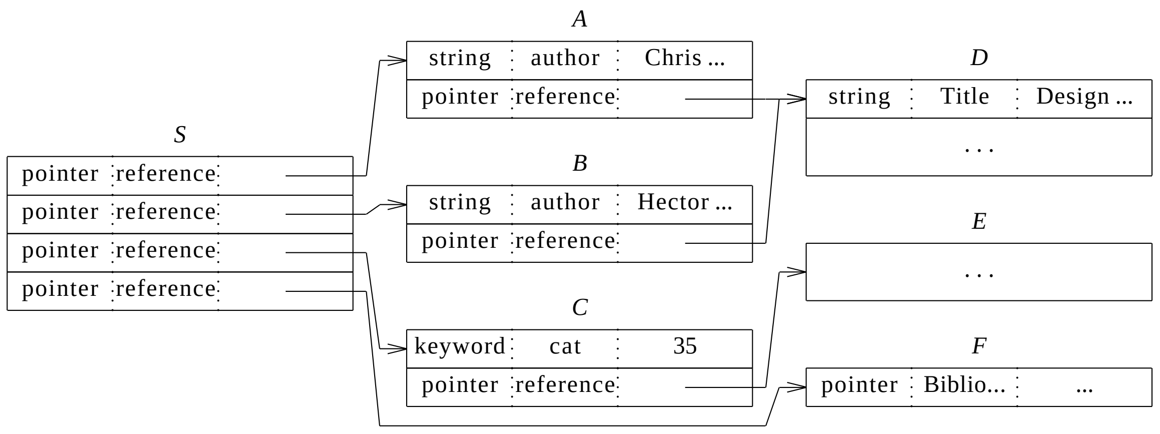


Figure 2.1: Set of documents.

- 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 described in Section 2.1 for single documents also have the appropriate meaning for sets of documents 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 that points to all of the objects in either *S* or *T*. In fact the primary use of these operations is likely to be on objects that are considered to be “sets of documents” rather than on individual items.

The basic filters of Section 2.2 select individual triples from an item based on the properties of those triples. With sets, we want to select objects from the set based on properties of the object pointed to. For example, in Figure 2.1 a query on set *S* would create a new set based on properties of *A*, *B*, *C*, and *F* rather than on properties of triples in *S*. This requires a different filter operation. These queries use the **|** operator, combined with filters that are similar to the basic filters discussed in Section 2.2. As an example, to select those documents from the set *S* that were written by either Chris or Hector we could use:[1]

S | **(** (string, "author", "Chris*") or (string, "author", "Hector*") **)** → *object id*

An equivalent statement would be:

((S | (string, "author", "Chris*")) ∪ (S | (string, "author", "Hector*"))**)** → *object id*

Note that we are using regular-expression style matching in the strings within a specific filter. This is a function of the underlying data type for that field. Chapter 7 contains a more complete discussion of the supported data types. Wild cards (**?**, described in Section 2.2) can also be used.

The **|** operator allows us to chain various filter parts. Each stage will “pass” only the objects which meet that particular criteria. The preceding example shows the *triple selection* filter, which selects objects containing the desired triple. (Following sections describe filter parts other than the triple selection filter.) One note: This Chapter gives an overview of the interface language, to serve as a framework for the rest of this thesis.

[1] The syntax used for our examples is what we use to represent the HIL. An implementation for a particular host programming language could use a different representation for HyperFile queries. What we are trying to describe here is the types of queries supported by HyperFile, rather than any specific representation of those queries.

Informal descriptions and examples are used to describe the semantics of the language. Those desiring a more rigorous treatment may wish to refer to Chapter 4, which describes the algorithm for processing these queries.

Here are additional examples of queries that use the triple selection filter:

Select objects in S with keyword "cat" and place them in document T.
    S | (keyword, "cat", ?) $\rightarrow$ T
Select objects with keyword having prefix "ca" .
    S | (keyword, "ca*", ?) $\rightarrow$ T
Select documents having keyword matching "?a?" with greater
    than 30% relevance to the object.
    S | (keyword, "?a?", >30) $\rightarrow$ T
Select documents with either "cat" or "dog" as a keyword.
    S | (keyword, "cat", ?) OR (keyword, "dog", ?) $\rightarrow$ T
Select documents having both "cat" and "dog" keywords.
    S | (keyword, "cat", ?) | (keyword, "dog", ?) $\rightarrow$ T
In the above query we could have used AND; using two filters has
    the same result (first select documents with "cat",
    then from that set choose those that have "dog".)
Select documents having "Princeton" as keyword or in the title.
    S | (keyword, "Princeton", ?) OR (string, "title", "*Princeton*") $\rightarrow$ T

2.4.1. Matching variables

Related to wild cards are *pattern matching variables*. These are wild card characters that must match at various points in an expression. For example the following query (on a sample software engineering database) chooses programs that are being maintained by their author:

    S | (String, "Author", ?X) | (String, "Maintained By", X) $\rightarrow$ T

In the portion of the query ... "Author", ?X) ... X becomes a set of all of the *Author*s of the object, and later these are compared against the values of *Maintained By* tuples **for that object**. If any of these matches a value in X the expression evaluates true and the program “passes” the query. Note that matching variables are used to compare values within a single object, not between objects.

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

    S | (String, "Author", ?X) | (String, "Author", X≠?) $\rightarrow$ 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 ? to the data field. Since X=?, the tuple does not match, and as there are no other author tuples the document 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 each name 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*) that is not *Chris*; this tuple matches. Since at least one tuple matches, the document passes the filter and is placed in the result set T.

The occurrence of the variable preceded by ? specifies that it is a binding occurrence, without the ? it tests for matches. Filters are evaluated left to right, hence the leftmost occurrence of a variable should be a binding occurrence (otherwise nothing will match.) Further binding occurrences add to the set of possible values for the variable for that object.

The actual semantics of pattern matching variables are similar to Prolog unification[Robi65]. The variables are bound to any pair of triples that may cause them to match. However, the filters restrict the scope of triples available for matching. This simplifies the problem of finding matches efficiently. Their use is also related to *joins* in a relational database. However, since matching variables only operate within an object the processing is simplified; “expensive” join queries do not exist. Another way of thinking of matching variables is that each instance of an object passing through a filter has its own matching variables. Each variable is a set of values corresponding to the bindings for that object. An expression using the variable is true if any of the values in the set would make the expression true. Chapter 4, which gives the query processing algorithm, formally describes how matching variables work.

2.4.2. Pointer operations

One of the defining features of HyperFile is the presence of pointers in the database. In order to allow following of pointers, two *dereferencing* operations are provided. These are $\uparrow X$ and $\uparrow\uparrow X$, where *X* is a matching variable. The first is a simple dereference; it returns the object pointed to. The second gives both the item pointed to and the original object. These are best shown by example:

S | (pointer, "reference", ?X) | $\uparrow$X $\rightarrow$ T_1

produces the items referenced by objects in *S*. T_1 is itself a set (object containing pointers), and can be operated on in the same manner as *S*. Using the set *S* in Figure 2.1, we see the query would result in T_1 containing *D* and *E*. If we wish to include the pointing objects in the result we use a $\uparrow\uparrow$:

S | (pointer, "reference", ?X) | $\uparrow\uparrow$X $\rightarrow$ T_2

returns the documents referenced to by documents in *S* and the referencing documents. Note that this is not all of the documents in *S*. The first filter removes documents that do not contain references. Using *S* from Figure 2.1, T_2 would be *{ A B C D E }*. Note that *F* is not in the set, as it does not contain a reference pointer.

This allows us to simulate the browsing model of hypertext by interspersing pointers with other types of filters. We can construct arbitrarily complex queries that look for objects meeting specific criteria, follow certain pointers from those objects, and so on. The set-based approach of filtering queries allows us to do this without the manual navigation and repeated user interactions of a standard hypermedia system. Additional examples of queries incorporating pointers:

Place documents bibliographically referenced by documents in S into T.
S | (pointer, "Bibliographic", ?X) | $\uparrow$X $\rightarrow$ T
Documents referenced with either Bibliographic or foo references.
S | (pointer, "Bibliographic", ?X) or (reference, "foo", ?X) | $\uparrow$X $\rightarrow$ T
References of documents with keyword "cat".
S | (keyword, "cat", ?) | (pointer, "reference", ?X) | $\uparrow$X $\rightarrow$ T
Documents that are references of documents in S, where the referenced document has the keyword "cat".
S | (pointer, "reference", ?X) | $\uparrow$X | (keyword, "cat", ?) $\rightarrow$ T

The dereferencing operations can be chained as well; we can follow pointers, look for selection criteria, follow other pointers based on these criteria, etc. This chaining allows applications to build complex queries out of a few “canned” application specific query parts.

Pointer operations can also be used with basic filters; for example

D (pointer, "reference", ?X) $\uparrow$X $\rightarrow$ T_3

produces the documents referenced in *D*. Note that T_3 is the union of the items referenced by D (that is, it contains all of the tuples of those items.)

2.4.3. Iteration

Sometimes we may want to follow pointers repetitively. To handle this, an *iteration* operation is provided. For example, we can find the papers referenced by those papers referenced by a given set *S* (two hops away from the given document) as follows:

S [| (pointer, "reference", ?X) | ↑X]^2 → T

The operations within **[ ]** are repeated as many times as indicated by the number given after the second bracket; the above statement is equivalent to:

S | (pointer, "reference", ?X) | ↑X | (pointer, "reference", ?Y) | ↑Y → T

Note that this is not quite syntactically equivalent; the matching variable *X* is rebound each time through the iteration.

When used with the ↑↑X operator, the query finds all documents **within** two hops (that is, the document, those one link away, and those two links away):

S [| (pointer, "reference", ?X) | ↑↑X]^2 → T

Note that we are apparently processing the original documents twice. In the first iteration, we find all of the documents one link away from those in the set. The second time, we repeat this, as well as finding documents two links away. Since the result is a set, the duplicates are eliminated. It should be remembered that this is a *semantic* model; the algorithm is clever in processing such a query and in fact only processes each object once.

If we want to find all objects within a tree rooted at the current set (transitive closure), we can use the ***** operation:

S [| (pointer, "reference", ?X) | ↑↑X]* → T

This repeats the operation in brackets until the result reaches a fixed point. At first glance, the following query would find all the leaves of the graph rooted at S:

S [| (pointer, "reference", ?X) | ↑X]* → T

This would return the empty set, however, as it would continue until there were no more referenced objects. All of the referencing objects would be thrown out by the dereference, the pointed to objects would be run back through the iterator and be discarded when dereferenced. If we wish to find all of the leaves we use the following:

S [| (pointer, "reference", ?X) | ↑↑X]* | NOT (Pointer, "reference", ?) → T

This gathers all of the items, and discards those that don't reference another (the leaves.)

2.4.4. Transferring Data to Applications

We have not yet shown how to actually manipulate the data items found with filters. This is because such manipulation is left to the application, and as such should be written in the language used to write the application. The data must be available to the host language, however. One way to accomplish this is simply to request an entire object from HyperFile (or part of an object, using a basic filter.)

Another method for transferring data to the application is with a method similar to pattern matching variables. Rather than the **?** used to set a matching variable, a → is used. For example →X (where X is defined in the host programming language) will set X to the values of the appropriate field. Note that multiple values for X may exist. In the Eiffel implementation, X is a set valued variable.

As another example, we could embed the HIL in C in a manner similar to QUEL[Allm76]. Using this method a section of code is executed once for each value, with the variable bound to a new value for each execution. This is probably easier to understand with an example. If we wished to display individually all of the titles of documents in written by Chris Clifton we could issue the query:

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

Note that these are exactly the documents in T (which is set immediately on return from the query, before the printfs are executed); T can be used as an initial set for further queries.

Here is an example of the same query using the Eiffel implementation. More code is required than the above C version, however this implementation does not require a preprocessor (as would the C version.)

```
hyperfile_server.send_query(
    "S | (String, \"Author\", \"Chris Clifton\") | (String, \"Title\", →title)");
result := hyperfile_server.get_result;      -- Note that send was non-blocking.
T := result.result_id;                      -- T is an object id.
title := result.result_table.item("title");   -- title is a LIST[STRING]
from title.start
  until title.offright
  loop
    putstring("Title "); putint(title.position); putstring(": ");
    putstring(title.item); new_line;
    title.next
  end;
```

The → operator may also be used with a basic filter, for example to retrieve the names of all authors of an object.

This gives an overview of the data model and query interface. The next Chapter gives a short comparison with other data management systems. Following that we will discuss a number of technical details and innovations of HyperFile.

CHAPTER 3

Previous Work

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

1. File Systems

HyperFile is probably most similar to a file system, particularly one with *self-describing data records*[Wied87]. In these systems records of a file contain tags stating what information is contained in the record, as opposed to either a heavily structured file (where each record contains the same type of information) or totally unstructured files.

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

File systems allow this flexibility, but provide little structure in places where it is desired. Items can be grouped in directories, and often hierarchical structure of the directories is allowed, but references and other pointers that are a part of many objects are not recognized by file systems. As discussed in the Introduction, file systems are inefficient for search and retrieval. In a large (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.

2. CODASYL Systems

HyperFile is similar to CODASYL[DBTG74] in that they both provide objects and pointers. A major difference between HyperFile and CODASYL, however, 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 Chapter 6), 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 that 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. Information Retrieval Systems

HyperFile is also similar to conventional information retrieval systems[Salt83] such as those for library applications. These systems allow filter queries similar to ours (e.g., find books with a given title), and indeed, our language was inspired by them. However, conventional information retrieval systems do not understand pointers. The ability to follow pointers within a query is essential to us, especially to support hypertext applications. Also, information retrieval systems typically do not support non-text data.

We view information retrieval systems as likely candidates for HyperFile applications. Ideas from these systems, combined with hypertext methods, can be used to form a general interface to a HyperFile database. Information retrieval research into automatic indexing[Salt88] and natural language[Crof87] can also be used to generate properties for textual objects.

4. Relational Systems

Relational systems[Codd70] provide a regular structure for data. This is both a blessing and a curse. Applications that involve homogeneous data often map nicely into a relational structure. These systems can then provide powerful queries on this data, as well as integrity constraints, transactions, etc. Not all data maps nicely into a relational structure, however. Although work has been done on placing text items in a relational database[Ston83, Smit86], creating a relational database that can support a variety of heterogeneous types of data is difficult.

HyperFile supports data that does not fit into a regular structure. Heterogeneous objects map easily into HyperFile objects. Another problem is that conventional relational systems do not support pointers; this is a serious shortcoming for our applications. Steps have been taken to address some of these problems in “advanced” relational systems (pointers, flexible data types, etc.), but we address these below.

5. Advanced Database Systems

Advanced database systems (such as object oriented[Maie86, Woel86, Wein88] and extended relational[Ston86, Schw86, Dada86]) provide many of the facilities of HyperFile (objects, pointers, queries), but also provide a lot more (like a full programming language or an inferencing engine). We feel that these systems may provide *too* much for a back-end document data server.

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

6. G^+

G^+ is a graph query language developed at the University of Toronto[Cruz87]. It has common goals with HyperFile, and provides a more powerful query language. Like HyperFile,

G^+ provides for graph based transitive-closure queries. Computing some G^+ queries can be NP-hard, however[Mend89]. This defeats our goal of providing a simple and efficient back-end data storage service. We have tried to keep our language simple, so that all queries will be computationally feasible. Our query processing algorithm (to be discussed in the next Chapter) is in fact linear in the number of objects processed.

7. Hypermedia systems

The initial motivation for HyperFile came from hypertext research. We view hypermedia systems as data *presentation* systems rather than data *management* systems. In other words, hypermedia systems are applications that will be used to access a HyperFile database.

Nevertheless, existing hypermedia systems do provide some data management features (perhaps due to the lack of a HyperFile data server.) It would be remiss to not discuss these. Most systems, such as KMS[Aksc88] Hypercard[Good87], and NoteCards[Hala87] provide little in the way of query facilities (beyond browsing, the shortcomings of which have already been discussed.) Some of these shortcomings, many addressed by HyperFile, are discussed in[Hala88]. Work has been done on improving the navigational aspects of browsing[Niel90], but this does not concentrate on the problems of scale addressed by HyperFile queries.

There has been work done on hypertext specific database[Tomp89, Smit86]. These only look at the data requirements of existing hypertext systems. HyperFile is a general purpose server for loosely structured data, and is intended to support a wide variety of applications in addition to hypermedia.

CHAPTER 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 Chapter.[1]

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

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

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

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

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

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

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

[1] We have not addressed query optimization. Some optimizations, such as reordering selections to perform highly selective operations early, may make sense in HyperFile. In some cases existing query optimization work may be able to be applied to HyperFile in a straightforward manner. We do address a related problem, indexing, in Chapter 6.

iterator I_1^3, which starts at F_1 and causes pointers to be followed for up to three levels. The last filter F_4 = (*keyword*, *Distributed*, ?) does simple pattern matching: Any object containing a tuple with type *keyword*, key *Distributed*, and any value for the data field will pass this section. The initial set S_i is S, and T will be bound to the result set S_θ.

Certain temporary information will be associated with each object O that 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.[2] An object is taken from the set and passed through the query from left to right. At each stage it can pass or fail to pass a filter, and may add new objects to the working set. At each stage the object is processed using the function E:

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

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

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

  The pattern may be a simple comparison (such as a regular expression

[2] 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. In any case the algorithm uses a node-based search. Work by Sarantos Kapidakis shows that a node-based search (processing each node entirely rather than, for example, following edges to new nodes before completing processing of the current node) will be fastest and require the least space in the average case[Kapi90].

for strings, or a range of values for a number). In this case matching involves equivalence of the pattern and the field in the tuple. The meaning of equivalence depends on the type of the field.

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

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

A matching variable may be used, as described in Section 2.4.1. 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 are not important to this discussion; matching is a function of the data type involved. Data types are discussed in more detail in Chapter 7.

```
$E(\ (type\_pattern,\ key\_pattern,\ data\_pattern),\ O)$ :
    for each tuple $t \in O$
        if $t.type = type\_pattern$ and
           $t.key$ matches $key\_pattern$ and
           $t.data$ matches $data\_pattern$ then
               $match = true$
               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 lines initialize $P$.
            $P.id = x$, $P.start = O.next + 1$, $P.next = O.next + 1$,
                $P.iter\# = O.iter\# + 1$, $P.mvars = \{\}$
            $Result\_set = Result\_set \cup \{P\}$
    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.[3]

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

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

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

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 4.1.) We will run the following query (described at the beginning of this section) on the set *S*:

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

The object *A* (the only thing in *S*) is processed. *A.iter#* is initialized to 1. In F_1 the matching variable X is set to the pointer (object id) *B*. F_2 dereferences this, setting *B.start* and *B.next* to 3, and *B.iter#* to *A.iter#*+1, or 2. The initialized *B* is then added to the set *W*. Next *A* continues processing with F_4, which checks for a keyword *distributed* and adds *A* to T if the keyword is found. Then *B* is then removed from the set, and starts processing at the iterator $F_3 = I_1^3$ (as *B.next* = 3.) Since *B.start* > 1 and *B.iter#* < 3 we realize *B* is new to the iterator and the result of a short chain of pointers, so *B* goes to F_1 (with *B.start* = 1.) Here X is set to *C*. In F_2 X is dereferenced; *C* is initialized with *C.start* = *C.next* = 3 and *C.iter#*=*B.iter#*+1=3 then placed in *W*. Next *B* reaches F_3, but this time *B.start* ≤ 1 so it continues processing with F_4. When *C* begins processing (at F_3) *C.iter#*≥3 and *C* exits the iteration (continuing with F_4.) Thus the query terminates before examining *D* (which is 4 levels deep.)

So far we have assumed that iterators are not nested. We do not expect nesting to be common, but it is handled with a slight extension to the above algorithms. The iteration number associated with an object *O* (*O.iter#*) is actually a 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

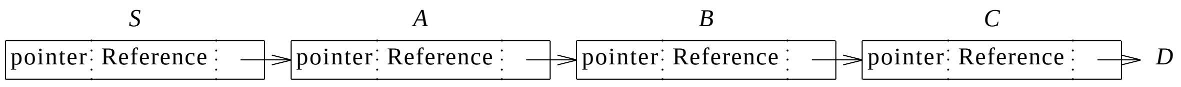


Figure 4.1: Chain of References.

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.

There is one important subtlety, however. Consider a query $Q = S_i\ F_1 F_2 F_3 F_4\ S_\theta$. Say a particular object O is in the initial set S_i, but fails to make it through filter F_1. Some other object containing a reference to O makes it through F_1, and in F_2 (a dereferencing filter) the pointer to O is dereferenced. Now we must realize that even though O was seen earlier (at F_1), it still needs to be processed starting at F_3. Thus, our mark table will record not only the identifiers of objects seen by a query, but also where in the query they were seen. In particular, $mark\_table(object\_id)$ will store a set of filter numbers. In our example, after processing O at F_1, $mark\_table(O) = \{1\}$. After O is processed at F_3, $mark\_table(O) = \{1, 3\}$. Figure 4.2 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 are the object id, iteration number and starting point in the query. This eases the task of parallel processing; to process an object in the set all that must be known is the original query Q, the information in the object O and the *mark_table*.

```
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 $O.next \le n$ do
            $mark\_table(O.id) = mark\_table(O.id) \cup \{O.next\}$
            $s, O = E(F_{O.next}, O)$
            $W = W \cup s$                           ;; add all dereferences to the set.
        If not null($O$) then
            $S_\theta = S_\theta \cup \{O\}$         ;;add $O$ to the result set
```

Figure 4.2: Query Processing Algorithm

CHAPTER 5

Distributed HyperFile

There are two main concerns in distributing HyperFile; how to describe references to non-local objects, and how to process queries that involve non-local objects. We will discuss query processing first. The implementation of remote pointers is discussed in Section 2. For now it is enough to assume that there is some way of mapping a pointer into both a location and the object at that location.

This Chapter also discusses some other issues involved in distributing HyperFile, as well as some experimental results from running queries on a distributed HyperFile server.

1. Distributed Query Processing

The filtering queries of HyperFile are simple to process in a distributed system. This only requires a slight extension of the processing algorithm presented in the previous section. The basic idea behind processing a reference to a remote site as part of a query is to send the query, not the data. The remote machine processes the query, and returns any results to the originating site of the query. We expect objects in our system to be long relative to the size of a query, so sending the query results in a considerable savings in communication cost over sending the unprocessed objects to the originating site. In addition, processing can continue at the originating site, taking advantage of the parallelism inherent in a distributed system.

Each site keeps a local context for queries it is processing. This context is a set of queries $\{Q_{1,}\ Q_{2,}\ \cdots\}$ where for each Q_i we have:

Q.id	An identifier for the query (assigned by the originating site.) Combined with *Q.originator*, this forms a globally unique identifier for the query.
Q.originator	The site that issued the query.
Q.body	The body ($F_1, F_{2,...,} F_n$) of the query.
Q.size	The length *n* (number of F_i) of the query.
Q.mark_table	The set of objects already processed (the *mark_table* described in the previous section.)
Q.W	The working set for this query.
Q.result	The set of results of the query.

A query is processed as follows:

- The originating site sets up a context *Q* for the query.
- The algorithm of Figure 4.2 is run, with the context *Q* used for the working set *W*, filters F_i, *mark_table*, and result set S_θ.

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

When site *R* receives the message, it tests if *Q.id@Q.originator* is already in its set of query contexts. If not, *Q* is added to the local query context, with *Q.result*, *Q.mark_table*, and *Q.W* set to *{}*. Then *O* is added to *Q.W*, with *O.next* set to *O.start* and *O.mvars* set to *{}*. If the algorithm of Figure 4.2 is not already running (that is, *O* is the only object in *Q.W*) it is started. Upon termination of the algorithm, *Q.result* is sent to *Q.originator*, and *Q.result* is reset to *{}*.

Note that after a site has emptied *Q.W* and sent results to *Q.originator*, another dereference message for *Q* may arrive. Since the context *Q* is still in place, the "setup cost" associated with the query is only required once at each involved site. The context *Q* is discarded only on global termination of the query (to be discussed in Section 3.)

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

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

- Object identifiers for objects that have passed all of the filters. These are put into the result set S_θ (*Q.result*) at the originating site. Further queries may use this set as a starting point (initial set S_i.)
- Tuple values returned using the $\rightarrow$ operator, as described in Section 2.4.4. These are sent to the originating site with a tag noting which $\rightarrow$ they belong to, so they can be bound to the proper variable in the application.

Cycle detection and marking are handled locally at each site. The information kept in *Q.mark_table* at each site refers only to objects processed at that site. If a site *R* has already processed an object *O*, and later another pointer to *O* is dereferenced, a message will be sent to *R* requesting that *O* be processed. Object *O* will be placed in the set *W* at *R*, but when it is removed from the set the "already processed" mark will be found in *Q.mark_table* and *O* will not be processed. Assuming data is not replicated, this is entirely adequate and prevents any repeated processing. Processing of replicated data will be discussed later.

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

Following is pseudocode for the distributed parts of the query processing algorithm.

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

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

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

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

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

Although we have covered the case of a distributed HyperFile server, it is important to note that our algorithms are also applicable to a shared memory, multi-processor server. In this case all available processors can share the same general query information, mark table, and working set. Each processor must have space for local information, such as matching variables, while it is processing a particular document. Given this, each processor independently runs the algorithm in Figure 4.2. Termination requires that the set be empty, *and* that no processors are still working on the query. Note that this is similar to processing of the Linda language[Carr86]. Also notice that it is not necessary to have a strict locking mechanism to prevent two processors from working on the same document. Duplicate processing may create some duplicate answers but not incorrect ones, due to the set-based nature of the result.

2. Distributing the Data

A major problem in creating a distributed database is **where** to place the data. This is highly dependent on the individual database, however, and must take into account the structure of the data, usage patterns, and possibly even legal and social issues (such as ownership of the data.) We can address the question of **how** to distribute the data. This issue deals with the mechanics of handling pointers to objects at remote sites.

2.1. Naming issues

Pointers form an important part of the database. Each object (set of tuples) has a unique *object id*, which can be used as a pointer to the object. Distributing the data requires a naming scheme so that these pointers can cross machine boundaries. There are two parts to this; ensuring that object identifiers remain unique, and translating pointers (object ids) into the site containing the object. For the latter we need some function $F(\text{Pointer}) \rightarrow \text{Location}$. There are a variety of ways to do this, such as global name servers[Birr82] or including the name of the host site as part of the pointer. There are a number of tradeoffs in the choice of a naming strategy:

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

The following paragraphs describe the naming strategy of HyperFile, as well as discussing some alternatives.

Name servers can add to the cost of dereferencing a pointer, particularly if the name server is at a remote site. The obvious alternative of including the host site as part of the pointer seriously increases the cost of moving an object, as all pointers to the object must be updated if it changes sites. We use a variant of the method of R^*[Lind81] that includes the *birth site* of an object in the name.

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

In a very large distributed system, the size of the birth site portion of the object id may be large. Storage space can be saved by abbreviating local references. For example, *object_A* located at *site_X* could have pointers to *object_b@site_X*, *object_c@site_X*, and *object_b@site_Y*. The first two could be abbreviated to *object_b* and *object_c*; the current site would default to *site_X*. HyperFile uses a variant of this; rather than abbreviating pointers to the current location we abbreviate pointers that have the same *birth site* as the pointing object. The above example shows the abbreviations used if *object_A* was born at *site_X* (that is, its full name is *object_A@site_X*) regardless of its current location.

The advantage to this method (over abbreviating references to the current location) is that abbreviations need not be expanded when an object is moved. Its birth site is unchanged, so the abbreviated pointers are still correct. In addition, many pointers in an object will refer to items with the same birth site (parts of a document, subroutines of a program, etc.) We expect more pointers will be abbreviated using our method than abbreviating references to the current site (for objects not located at their birth site.)

We may also desire hierarchy of names: In addition to local and global names, we may have “cluster” names that are good within a small subset of the system (such as a single organization.) References within the same cluster could be abbreviated. This is not implemented in HyperFile, however.

Another issue is foreign references: Those to objects not in the HyperFile database. For example, this thesis contains a list of references. Many of these would not be available in a “Princeton-wide” database. We would like to be able to handle these references in as close a manner as possible to normal pointers. This is done using *stub documents*, which contain bibliographic information. Information on where to find the desired object would be provided in lieu of displaying the object itself. This can be done entirely by the applications, with no special treatment by HyperFile.

2.2. Distributing within an Object

In some cases it may make sense to place different parts of an object at different sites. For example, one machine may be optimized for storing and processing video, where another may be useful for keyword and text searches. In such a case, an object containing both types of data would best be stored at multiple sites. We currently handle this by making a separate object at each site. The applications are responsible for handling the distribution. Making this type of distribution transparent to the applications is an area for further research. This would require some special handling in query processing; possibly "active objects" that when queried would trigger queries at remote parts of the document.

3. Query Termination

With only a single site, a query terminates when its working set becomes empty. With multiple sites, however, all of the working sets must be empty. Determining when this has happened is an instance of the *Distributed Termination Problem*[Fran80], which has been the subject of considerable research.

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

An obvious solution is to have all sites report completion to the site (or sites) that sent them a piece of the computation. Each message (pointer dereference) generates a *task*; the task is complete only when all local processing related to the task is complete, and all subtasks at other sites generated by processing of the task have reported completion. This generates a "tree of tasks", and requires that considerable state information be kept to determine if a task has finished, or is waiting for more tasks to complete. A single site may contain multiple tasks, for example if site *A* sends a reference to site *B*, and site *B* later sends a reference back to *A*. Site *A* must tell *B* that the task from the second message is complete before *B* can report that the task from the first message is complete.

With this method the number of messages sent for each query is doubled; each outgoing message requires a return completion message. A larger difficulty with this solution is the time required; the total delay required to detect termination from the time that termination occurs may be $O(m)$, where m is the number of messages sent in the original computation.

More efficient algorithms have been developed. These require less time, less local state information, and in some cases fewer messages than the above solution. We plan to use the weighted messages algorithm[Huan89, Roku88]. This algorithm works as follows:

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

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

In our HyperFile implementation, there are two particular issues that must be covered:

- Precision (how to divide weights when sending messages.) With a fixed-length representation of weight, a site can be left with the smallest possible unit of weight. The site would then be unable to send messages. This is unacceptable for our implementation. Existing versions of the weighted messages algorithm either ignore this problem[Huan89] or propose a mechanism that generates more messages (request additional weight from the originating site[Roku88].) Our solution is to use variable-length encoding; weights are sent as rational numbers, with the numerator and denominator stored as binary integers. This allows *infinite* division of weight, at some expense in message length. This shouldn't be a problem in practice -- reasonable choices of how to split weights (based on the expected number of messages to be sent) will limit lengths to a few bytes.
- Too many control messages. Every time a site empties the queue of objects to be processed for a query, it sends a control message to the originating site. It may then be restarted if another message is received. This is inefficient if sites become idle and are reactivated frequently. To handle this, each site can delay for some δ time units before sending the *done* message. This would allow for replies to be "batched together". This length of δ would have to be determined by simulation or experience with the system, and would reflect the expected amount of time between a processor emptying its queue and a new reference arriving.

4. Reliability

There are two reliability issues specifically related to distributing HyperFile. One is the problem of maintaining data integrity in the face of media failures; replication can be used to help this. Another issue is *availability*; allowing queries to progress in the event of failure.

4.1. Replication

A distributed database allows us to ensure the overall integrity of the database in the face of media failures by keeping copies of the data at different sites. Replication can also improve query performance by allowing queries to access the “closest” copy of a data. This is a well-studied problem[Mahm76, Elli77, Garc81]. We present one scheme that is appropriate for HyperFile, and fits nicely with the query processing algorithms (although we do not rule out other methods.)

Replicated data could cause inefficiencies in both storage space and query processing time, depending on how replication is implemented. One option is for applications to handle replication; an object would contain tuples with pointers to all copies of a referenced object. The problem with this is that queries will follow all of these pointers, instead of querying a single copy. This is unnecessary. In addition the storage space requirements increase, as pointers *to* replicated objects are replicated as well.

A similar solution is to store multiple pointers to a replicated object within a single tuple. This would be invisible to the application. Pointers to a replicated object could be dereferenced in a standard order, so only one copy of each object would be processed. The underlying communications protocol would have to notify HyperFile if a message could not be delivered, so the failed dereference could be sent to the next copy. This still has the problem of requiring storage space for the extra pointers.

We use a variant of this solution. Each reference only contains a single pointer. A site caches one (or more) current locations for an object. If this site (or sites) has failed, the message is sent to the *birth site* of the object. The birth site is responsible for knowing of all copies of the object. Note that this is the same action taken if the sender did not know the current site of the referenced object to begin with. Updates must all pass through the birth site to ensure reaching all copies (an expensive solution, but cost-effective given a high read to write ratio.) If the birth site is lost, it can be reconstructed by broadcasting a request for the location of all copies of an object. If this is too expensive, backups for the birth site may be kept (these would act as *surrogates* if the birth site is lost.)

This method allows for a variety of tradeoffs. Reliability can be improved by making more copies of an object; this also gives potential for improvement when reading the object but increases the expense of updating the object. Note that this choice can be made on an object by object basis. Another advantage to this scheme is that no global control is required; the birth site of an object is in effect the *manager* of that object, and can make decisions as to what degree of replication is desirable. This autonomy may be particularly appropriate in widely distributed databases that span multiple organizations.

It is okay for different pointers to an object to be translated into different copies. This encourages use of local copies of a replicated object. In some cases this may cause inefficiencies, however. For example, during a query a pointer in object *A* is followed to copy 1 of an object *C*, and a pointer in *B* is followed to copy 2 of *C*. Both copies of *C* are processed. Any pointers from *C* are followed twice, requiring extra messages. Happily this is only a minor problem. Even if a replicated object is processed at multiple sites, the results obtained from each site will be the same. The originating site eliminates the duplicate results.

Allowing this redundant processing of replicated copies will cause some increase in message traffic over non-replicated data. The references from *A* and *B* would still have required messages to the same (single) copy of *C*, so this portion of the traffic will not increase. However, messages generated by following pointers from *C* would be duplicated if two copies of *C* were processed. The end result is that replication will often decrease the time required to obtain first results, but time to termination may be increased. No changes to the algorithms are required, other than message forwarding by the birth site. If a message for a remote reference cannot be delivered, it will be sent to the birth site. The birth site will attempt to find an accessible copy of the object.

4.2. Availability

Availability concerns keeping the database usable in the face of failures of part of the system. In our case we are interested in failures of some of the sites that contain objects needed as part of a query. One of the issues of increasing availability, making objects available even if the machine they are on fails, is handled using the replication scheme of the preceding section. If the cached current site (or sites) is not available, the message is sent to the birth site. Although this method does not tolerate failures of all of the cached current sites and the birth site, each site can decide how much space (in terms of caching additional current sites) it is willing to use to increase fault tolerance. Handling more failures without expanding the space requirements of the cache would require either some sort of a global locating service, or broadcasting the request for the object. The former gives up some of the independence and autonomy of our scheme, while the latter could be too expensive in a widely-distributed system where the number of copies may be large and the time for a broadcast considerable.

Another issue is what to do with a reference to an object that *cannot* be reached. Related to this is what to do with a query that encounters unreachable references. We report pointers that cannot be followed to the originating site. This leaves the originating site with two options: Abort the query, or report the problem to the application. The latter solution allows the application to decide if an abort is necessary. The query will report results received, as well as giving a list of unreachable references. For some applications, such as a looking for specific information on a topic, this may be adequate. For example, a query might ask for specifications on a VLSI chip. This could return many kinds of information; timing diagrams, pin-outs, power requirements, etc. The desired information may be contained in the available results, and the user could proceed even though part of the query could not complete.

A more serious problem is what happens if a machine fails while it is processing a query. The distributed termination solution outlined in Section 3 is not particularly robust. In particular, if a site fails while it still has some weight *w*, that weight will never be returned to the originating site. Thus the query will never terminate. HyperFile times out if the query seems stalled, and reports partial results to the application in the same manner as with unreachable pointers. More robust distributed termination protocols exist[Lai86], but they are also more complex and expensive. These could easily be placed in HyperFile in a distributed system where the timeout method is inadequate.

5. Costs

We have said that this is an efficient means of processing distributed queries, to justify this we will discuss some of the costs of processing distributed HyperFile queries. We will examine two types of costs: The increase in local processing cost, and the generated message traffic.

As discussed previously, the local cost of **sending** a remote query is insignificant: The query and associated *fixed* information is only built once at the originating site, and then the message is reused for all further remote queries. The remaining cost in building the message; the reference, point in the query, and iteration information; must be determined in order to place in the queue for a local query. The only difference is placing it in a message to be sent rather than in the working queue. **Receiving** a remote query involves some cost: The local data structures to hold the query and working queue must be built. This need happen only once at each site, however. The cost per-reference is only putting the reference information into the queue. This is comparable to the cost required to enqueue the reference if it were local.

Another cost that cannot be ignored is the translation from the global naming scheme into local names. This, however, is inherent in the distributed system and not due to the query processing algorithm.

Communications costs can be significant. Each message is small (except possibly results, but that can't be helped), but there may be many messages. The messages are divided into two types: Pointer messages, and control messages for the termination algorithm. The number of pointer messages can be as large as the number of items processed by the query. Alternative algorithms could store messages and send them as a group, cutting the number of messages to something on the order of the number of sites involved. This would complicate query processing, however, and we feel that it would not be necessary in practice. Unless some special information about the global structure of the database is kept at each site, the number of pointer messages is worst-case optimal. A simple demonstration is to imagine a traversal of a linked list, where each **odd** element of the list is at the originating site, and each **even** element is at the remote site. After the first document is processed, nothing can happen until a message is sent to the remote site (as the original site doesn't know that it has more documents to be processed locally.) The remote site can either process the second document, or send it to the original site. The latter case costs a message immediately, but no further message is required to process the third document. The former case delays the cost of this message until the second document is processed, but it is still required.

This worst case demonstration isn't all that applicable in practice. It is easy to imagine cases where our algorithm is *not* optimal. For example, complete processing of the queue on the original site could cause numerous references to a remote site. Under our algorithm, these would be sent individually. These could be bundled and sent as a single message. This could be incorporated into our algorithm, but the decreased message cost would also result in decreased concurrency. This is a tradeoff that could be figured out for each system.

The cost of control messages (to detect termination) ranges from the number of pointer messages to the number of involved sites, depending on how the termination algorithm is run (as described in Section 3.) Other algorithms could possibly eliminate this cost or assume it entirely within other messages. However, this would increase the complexity of the algorithm. We feel the cost of the potentially large number of messages is a worthwhile tradeoff for the simplicity of the query processing algorithm.

A broadcast from the originating site to all sites involved in the query is also needed once termination has been detected. This allows the remote sites to forget the processed object mark table and other information pertinent to the (now complete) query.

6. Experiments

One of the advantages of the distributed query processing algorithm is that it needs little central control. The downside to this is that it is difficult to predict just how it will perform. As a result we have run some experiments to test how HyperFile operates under various situations.

We have implemented this algorithm in a prototype HyperFile server, distributed over a network of IBM PC/RTs connected by an ethernet. The RTs run Berkeley 4.3 UNIX; UDP and TCP/IP are used for inter-process communication.[1] Each machine has a single server. This is a main memory database (as will be described in Chapter 7); although large objects are stored on disk none of our test queries required disk access. The implementation is not particularly efficient; we have concentrated on extensibility rather than speed. An

[1] This implementation was done using the Eiffel object-oriented language. The version of Eiffel used (v2.2) did not support inter-process communication; we created a message based system for communicating Eiffel objects in order to support this work.

optimized system would significantly decrease the times we present. Our experimental client was a simple application that read a query from a script, submitted it to HyperFile, received the result, and then went on to the next query in the script. The client ran at a separate machine from any of the servers.

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

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

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

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

- Five **search key** tuples; one guaranteed to be unique to that object, one found in all objects, and three that were chosen from a space of 10, 100, and 1000 possible values respectively. Changing the tuple and value searched for allowed us to vary the number of items found by a query. For example, searching for a given key in the *unique* tuple would return at most one object.
- One **chain** pointer that gave a linked list of all the items. In tests with more than a single machine, these pointers were always to a remote machine. This gives the maximum *delay* time; all servers are idle while each message is in transit.

- Fourteen **random** pointers. These each pointed to a randomly chosen object. They were divided into 7 types, with two pointers of each type. The probability of a pointer being to a local object varied from .05 to .95 depending on the type. For example, the two pointers of the **Rand.05** type were almost always to a remote object. A query following the Rand.05 pointers would have high message cost. However, since there were two such pointers in each object (very likely to different machines) the query would "branch out", yielding some parallelism and reduced delays.
- **Tree** pointers that formed a spanning tree of the objects, such that the root of the tree had a single remote pointer to all other machines, and each of these was the root of a local spanning tree. This gives high parallelism with low message cost.

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

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

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

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

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

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

There were 270 objects involved in the queries for which we report results. (Note that the total database was larger; however only 270 objects were looked at by our test queries.) As the algorithm is linear we expect using a different number of items in the query would result in a linear change in the response time. We did construct a data set with half the number of items; this didn't quite cut the query time in half. This is as we would expect (since there is some constant overhead associated with the query, regardless of size.) Presenting more experiments with varied data set sizes would tell little of interest; our primary concern is how remote pointers affect performance.

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

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

The above two cases are extremes. To study "normal" situations we ran tests on the randomly constructed pointers. Although still synthetic data, they are probably more representative of real situations. The results of these tests are graphed in Figure 5.1. Each data point represents a test using the graph formed by the pointers with the given

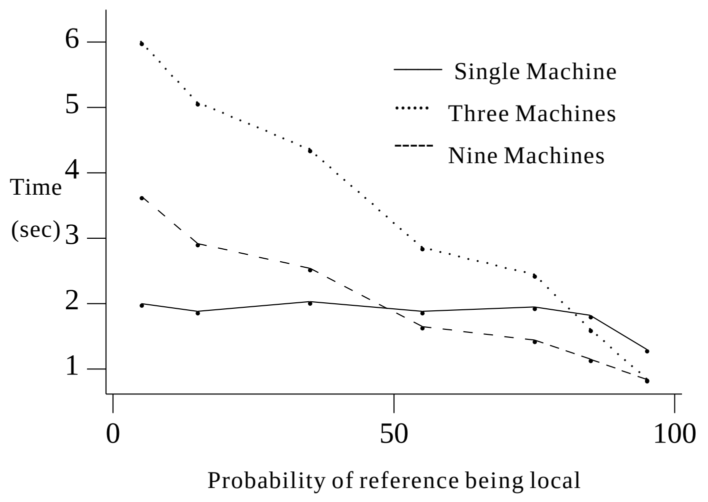


Figure 5.1: Query time with increasing probability of local references.

probability (*x axis*) of being local (two such pointers per object.) The cases at the far right of the graph generate fewer messages, however they also are less likely to make full use of the available parallelism. The cases at the far left generate too much message traffic for our system; although parallelism is increased, much of the time is spent receiving and sending messages rather than processing queries.

It would be reasonable to expect that the single-machine case would be constant. This is not the case. The reason is that the pointers in these tests were created randomly (within the local/remote guidelines), and the transitive closure of a given pointer type was not guaranteed to include 270 objects. The single machine case gives a measure of the number of items actually covered, so it is perhaps more relevant to look at the difference between the dotted or dashed line and the solid line, rather than the absolute times. This has been done in Figure 5.2; here we see the relative speeds of the various cases. The points are the actual data, the lines are a linear least-squares fit to the data points. The best case would have the three and nine machine cases approach 1/3 and 1/9, respectively (representing perfect speedup as communication costs go to zero.) As we decrease the number of remote references, however, the likelihood of machines sitting idle increases, which is why we do not achieve perfect speedup. This implementation achieves the best response time with over 90% local references. An implementation with a more efficient

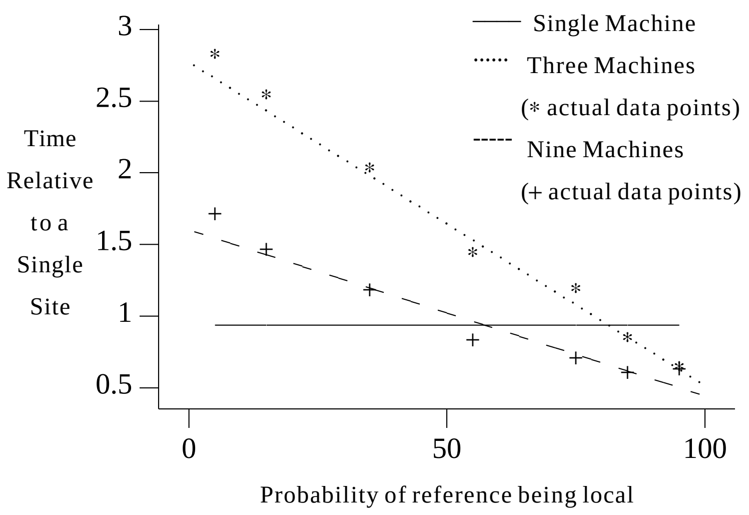


Figure 5.2: Query speedup with increasing probability of local references.

communication system (relative to processing cost) would achieve its best response time with a lower percentage of local references.

The user response time is actually improved in distributed HyperFile as long as most references are local (a reasonable assumption.) We also see that with more machines we are capable of handling a higher percentage of remote references. This is good, as a more highly fragmented database will probably have more remote references.

Another interesting result concerns the number of items returned by a query. Increasing the number of items returned significantly increases the query processing time. Given two queries that follow the same pointers, a highly selective query may be faster in the distributed case, but a less selective query may run faster when the entire database is on a single server. For example, the case in Figure 5.1 where 95% of the pointers are local takes an average 1.1 seconds when run on three or nine machines, and 1.5 seconds when run at a single site. Note that this is returning an average 10% of the items in the transitive closure. If we instead select all of the items (using a key that is found in all of the objects) the single site time jumps to 5.1 seconds. For three and nine sites we have 6.4 and 5.7 seconds. This is illustrated in Figure 5.3. Sending results is expensive in our system; we would have to make changes if queries with low selectivity are frequent. We expect this will not be the case, as the goal of most queries is to find a *few* interesting objects.

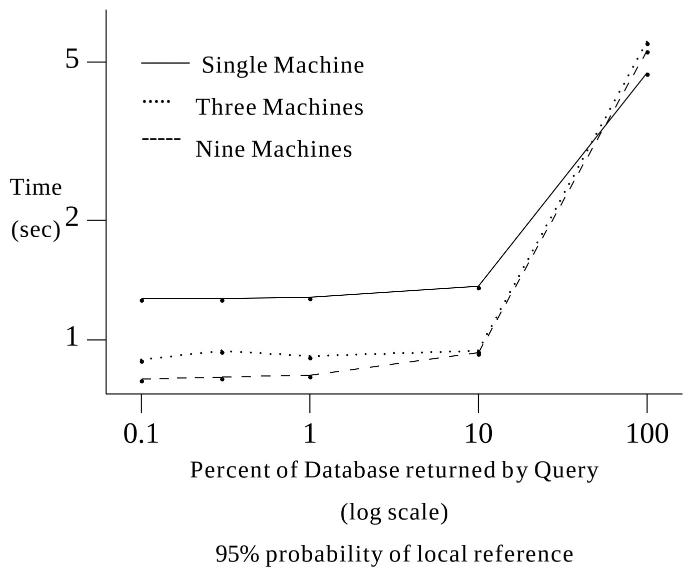


Figure 5.3: Query time with varied number of returned results, 95% local references.

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

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

CHAPTER 6

Indexing

As with many large databases, some HyperFile queries can take considerable time to process. A query that searches every item in the database can take time that an interactive user would consider unreasonable. Indexing is commonly used in traditional databases to speed up these searches by effectively "precomputing" parts of common queries. We use indexing in HyperFile for the same reason.

Indexing in HyperFile demands some new 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. 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 that could be reached from that node following a particular type of link in a "browsing" interface (the *scope* of the query.)

The indexing methods described here may have applications other than HyperFile. In particular, any transitive queries on hypertext-like data may benefit from this work. There may also be applications in object-oriented database. This is an area for further study.

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 that could be reached from that node following a particular type of link in a "browsing" interface (the *range* of the query.)

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. Specifying the *scope* of the index, however, is different. Rather than specifying a *relation* or *set* that 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 will thus require specifying three parameters: The *anchor point* (node) that the index is to be connected to, the *search key* for the index, and the *link type* that determines the scope of the index.

Figure 6.1 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 by a dashed link.
- Item *I* is pointed to by a solid link. Since it is not **reachable** from *root* via solid links, however, 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 6.1 will speed up searches whose scope is the solid-link tree rooted at *root*. The Database Administrator is the one that determines that such an index is useful, based on the expected queries. The DBA has much the same responsibility in a relational

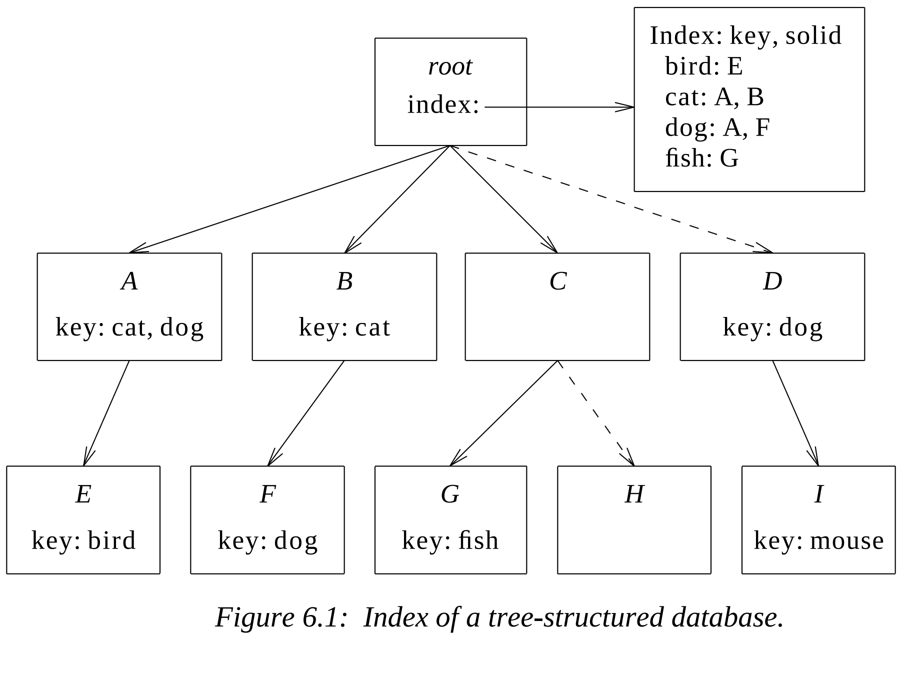


Figure 6.1: Index of a tree-structured database.

system.

2. Structure of the index

The index itself will be structured in a similar manner to a traditional database index. B-trees, hashing, and other such techniques are all applicable. Certain special information is required, however. In addition to pointers from the index to relevant objects, objects will be required to have back pointers to indexes that *potentially* include them. This is necessary in order to properly maintain the index. For example, in Figure 6.1, *C* will have a back-pointer to ensure that updates that add keys to it 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 pointers to the index in the parents of the links will point to the need for including *D*, *H*, and *I* in the index.

In a relational database, information about what indexes may potentially reference a given record can be determined easily from the definition of the index, due to the static nature of the scope of the index. In HyperFile system, determining what indexes a data item is in may be as difficult as building the index (in terms of number of items referenced), as the scope is determined by information contained in the objects themselves (pointers.) We use back-pointers to cut the costs of maintaining the indexes when data items are modified. In addition, when a data item is added to the database the indexes that refer to it can be determined from the index links of the parent of the item. We also need back pointers from all nodes in the scope of the index (even if they are not in the index, such as *C* in Figure 6.1) to support deletion. Deletion of a node or link may require changes in the index to deal with nodes below that point.

Following are pseudo-code descriptions for the various operations relevant to indexes. These will work only on tree-structured databases; the extensions necessary to operate on arbitrary databases will be discussed later.

```
Create_index ( node, key, link )
    Create an empty index data structure.
    Add a pointer to node noting the presence of the index.
    add_index ( index_structure, node, key, link )

Add_index ( index, node, key, link )
    Add all appropriate key items of node to index.
    ∀ children of node via link
        add_index ( index, child, key, link )

Find ( node, key_type, key_item, link )
    if node has a pointer to an index on  key_type and link then
        index_find key_item
    else
        if key_item present at node then
            Result := node
        ∀ children of node via link
            Result := Result $\cup$ Find ( child, ... )

Add_link ( parent, new_node, link )
    Add link to the database in the normal manner.
    ∀ index back-pointers in parent
        if index.link = link then
            add_index ( index, new_node, index.key, link )

Delete_link ( parent, child, link )
    Delete link from the database in the normal manner.
    ∀ index back-pointers in child
        if index.link = link then
            delete_index (index, child, index.key, link )

Delete_index is analogous to add_index
```

Searches from a node that is not indexed can still make use of indexes. The simple case is making use of an index that is associated with a node that is reached at some point in the search. This is already done in the above algorithms. In some cases it may be worthwhile to use an index located above the start point of the search. If the start point is in the scope of the index; the index will cover a superset of the desired search. Such an index can be found because the starting node of the search will have a back pointer to the index. All of the items returned by the index must be checked to see if they are in the proper subtree. For example, in Figure 6.1, a search from node *A* could use the *root* index, and then check all of the objects found by backtracking from the object until either *A* or *root* is reached. This assumes that the database provides back-pointers for all links. In many cases this may be done for reasons independent of indexing.

This is an appropriate approach when few items are found in a search of the index, and the subtree rooted at the search node is a large fraction of the subtree rooted at the indexed node. The given example would be slower than a direct search for the keys *cat* and *dog*, but would be comparable given a search on *bird*.

Determining when to use an index located above the search point is a difficult problem. Some simple heuristics that suggest the use of such an index are:

- The index returns a relatively small number of items compared to the size of the subtree to be searched.
- The desired subtree is a large fraction of the total size of the indexed subtree.
- The subtrees are relatively *broad*; back searches will require tracing a small number of pointers relative to the size of the subtree.

Even if we do not use an index above the start point of the search to actually find the desired objects, it may be of some use. If an index lookup returns no items for the desired key, we know that the search would also return an empty result (since the index covers a superset of the portion of the database being searched.) If searches often come up empty, this will result in a net savings.

3. Multiple Indices

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 6.2 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*. In order 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 that is below *A* must have back-pointers to both indexes.

4. Chained Indices

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

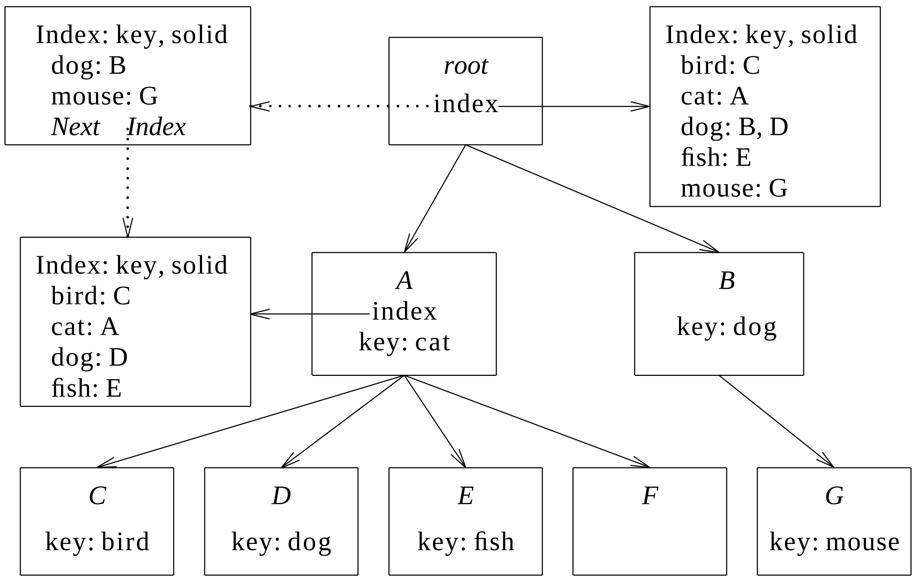


Figure 6.2: Tree-structured database with two indexes.

indexes.

This replication can be eliminated by requiring indexes to refer to “lower” indexes, rather than directly indexing the entire subtree. This is illustrated by the indexes pointed to by **dotted** lines in Figure 6.2 (just the ones on the left side of the Figure.) 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 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. We expect the typical cost will be much smaller, however. This will be discussed in Section 6.

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.

In some cases partial redundancy can be allowed. For example, if a new index is created beneath an existing one, the redundant items need not be immediately removed from the old index. This speeds the creation of the new index. The old index need be modified only when objects it indexes are changed. These data items will already have pointers to the old index. These pointers must be changed to reflect that updates to these data items should cause them to be removed from the old index. Changing the pointers, however, can be done as part of the creation of the new index. This adds only a constant factor to the time required to build the new index. This is one example of the numerous time/space tradeoffs that can be made with this indexing.

Eliminating replication may help when using indexes located *above* the start point of the query. For example, in Figure 6.2 a search from *B* could use the index at *root.* A clever implementation could note that the non-replicated index at *root* (pointed to by a dotted line) indexes *root* + *the tree rooted at B* − *the tree rooted at A.* This is very close to an index on *B.* A search from *B* could just use this index, and remove *root* from the result set.

5. 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[Lum70]. 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 that 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, sorted lists, etc. A naive implementation of the secondary indexes, where 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 6.2 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 point only those objects for which it is the "closest" anchor point, cutting the space considerably (worst case $O(n)$.)

For example, Figure 6.3 is a sample index containing entries for a few keywords based on the database of Figure 6.2 (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* that 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. More formally, the **Find** procedure is:

```
Find ( node, key_type, key_item, link )
    S = find_secondary_index ( key_item, link )
    let T = transitive closure of node in reachability graph for link

    Note that the previous two steps can occur in parallel.

    ∀ anchors A in T
        Result := Result ∪ S(A)  (Objects in A in secondary index T.)
```

As written this assumes that the current node has an index. Extending it to the general case is straightforward, and can be seen from looking at the Find operation of Section 2.

Lookup time for the reachability graph (finding transitive closure) is worst-case linear in the number of anchor points. Improving this time requires precomputing the transitive closures, which could take quadratic space (and is also expensive to compute[Ullm90].) There are better options, however. A number of transitive closure algorithms suitable for our secondary indices are given in[Jaga90]. A technique for a tree-structured graph (or

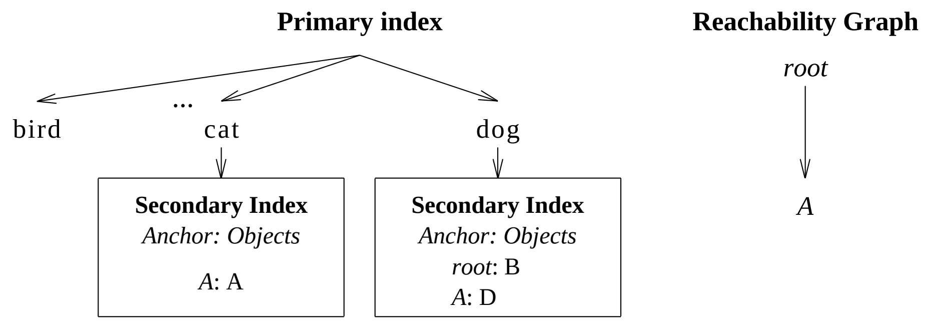


Figure 6.3: Single Multiple-Attribute Index.

tree-structured parts of the graph) expresses reachability as a *range of integers*. To do this, we name the anchor points by preordering the tree. With each anchor point, we store its number and the number of its right sibling. From a node *i* with right sibling *j*, the reachable anchor points are those numbered *i* to *j-1*. This cuts the "transitive closure" operation on the reachability graph to constant time with space linear in the number of anchor points.

Up to this point we have ignored different link types. Using the methods of the previous sections we have to construct a new index for each type of link. With this method we may reuse the primary index, however. Each key value will have a different secondary index for each link type, and there will be a separate reachability graph for each type of link. Also note that an index on a different key attribute can reuse existing reachability maps.

Updates to the database that change the key attribute of a data item will require that it be moved to a new secondary index. This requires no extra links; Finds in the primary index can be used to return the old and new secondary indexes. The node is then removed from the appropriate anchor point list in the old index, and added to the list for the same anchor point in the new index. Deletions and additions are similar. Changes to links are somewhat more difficult. For this we still need the back pointers from nodes to anchor points (as in Sections 2, 3, and 4) and from anchor points to the secondary indexes. Deleting or adding a link will require modifying some of the secondary indexes, and in some cases may require rebuilding part of the primary index (for example, if a new value for the key attribute appears.) In addition, the reachability graph may have to be changed.

6. 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. A simple estimate of the time and space costs for each technique on a regularly-structured database is given in this section. This provides for a reasonable basis of comparison of the indexing methods.

First we will set out the assumptions and terms used in these calculations. Although the techniques work for an arbitrary directed-graph structured database, we continue to 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. The analysis for using indexes located above the start node is too complex to present in detail here.

For the purposes of this discussion we will assume that the data and pointers to be 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[Ande89] uses such an arrangement as one of its three "hierarchies". In the next 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 2. Allowing a more varied placement of indexes increases complexity significantly; we felt the knowledge gained would not justify the increased effort. We need to define the parameters that we will use:

$T(n)$	Time required to do an index find operation on an index containing n elements. This will typically be logarithmic, and is determined by the choice of index (B+ trees, tries, etc.)
$E(n)$	Time required to search through n nodes without using an index. This will basically be linear, although the function could be complex if the data items are stored on disk.
c_s	Space required to store a search key in an index. Some index structures, such as *tries* or C_0 trees[Orla88] do not require linear space for the keys. Such structures would complicate this analysis considerably, but would be of most benefit to the single multiple-attribute indexes.
c_p	Space required to store a pointer in an index.

c_r	Space required for each item in the reachability graph of the single multiple-attribute index described in Section 5.
t_r	Time required to lookup an item in memory, such as in the reachability graph or in a linear search of the secondary index.
K	Total number of possible search keys.
P	Probability that a given key attribute value appears in a given data item. KP gives the expected number of key attributes per data item.
B	Branching factor. This is the number of children of any given data item (except for leaf nodes.)
j	Depth of the second (non-root) layer of indexes. The total depth of the tree is $2j$. We will consider *root* to be at level 0, and the leaves to be at level $2j-1$.
N	Number of indexable items in the database. This is equal to $B^{2j}-1$.

Note that there are B^j second level indexes. Each of these indexes has B^j-1 data items located beneath it. We have not put in a separate space cost for back-pointers from data items to the index. There will be one such pointer for every pointer from an index to a data item, so this is included in c_p.

We will use three queries in this analysis, each reflecting a different **start point**. From these three, we can predict results for queries from any start point. The queries are:

F_1	Find time for searches starting at the root node (which contains an index.)
F_2	Searches starting at a child of the root node. These will progress through half the depth of the tree before they are able to use second level indexes (if any.)
F_3	Finds starting at level j. These will be able to make use of a second level index directly (if one exists.)

Note that searches starting from below level j (below F_3) will take the same time for all of the methods, as no index will be used. Searches from between level 2 and j will take between F_2 and F_3 time, but will vary at the same rate for each of the three indexing techniques. We will use F_{it} to denote the time required for search F_i (where i is 1, 2, or 3) using index type t (where t is s for a single index at root, r for fully replicated indices, u for unreplicated (linked) indexes, and m for the single multiple-attribute index.)

As a quick example, for a single index located at *root* we have $F_{1s} = T(k)$, where k is the number of keys in the index; plus the retrieval time $E(r)$ for the r items found by the index. Given K total possible keys, P probability that a given node will contain a given key, and N nodes, we can see that the expected number of keys in the index (k) is:

$$k = K(1-(1-P)^N)$$

keys. The expected number of items to be retrieved r is PN. Therefore the expected retrieval time for a search from *root* is:

$$F_{1s} = T(K(1-(1-P)^N))+E(PN)$$

Searches from below the root require searching the entire subtree from the start point (which includes the object retrieval time):

$$F_{2s} = E(B^{2j-1}-1)$$
$$F_{3s} = E(B^j-1)$$

As to the space requirement, note that an index will require $c_s k + c_p d$ storage space, where k is the number of keys in the index (as determined above), and d is the number of items indexed. Also, a given database item will have pointers to it in the index KP times, so we have an expected value for d of nKP. This gives us a storage space requirement for an index of size n of

$$S(n) \quad = c_s K(1-(1-P)^n)+c_p nKP$$

Therefore the space requirement for a single index at *root* is

$$S_s(N) = c_s K(1-(1-P)^N)+c_p NKP$$

Using multiple indexes without eliminating replication gives the fastest lookup time of any of the three indexing methods described. Starting at the root we get:

$$F_{1r} = F_{1s} = T(K(1-(1-P)^N))+E(PN)$$

If we start at level 1 things are somewhat worse. We have to first search all of the nodes between the start point and the relevant second level indexes ($B^{j-1}-1$ nodes), and then use each of the indexes beneath this point.

$$F_{2r} = E(B^{j-1}-1)+B^{j-1}T(K(1-(1-P)^{B^j-1}))+E(P(B^j-1))$$

Finally, at level j we need search only a single index on B^j-1 items:

$$F_{3r} = T(K(1-(1-P)^{B^j-1}))+E(P(B^j-1))$$

This method requires the most space. To the space requirements for the single index we must add B^j smaller indexes at level *j*. Thus the total space requirement for the replicated multiple index technique is:

$$\begin{aligned} S_r &= S(N) + B^j S(B^j - 1) \\ &= c_s K(1-(1-P)^N) + c_p NKP + B^j K(c_s(1-(1-P)^{B^j-1}) + c_p P(B^j - 1)) \end{aligned}$$

Eliminating replication saves space at some expense in time for searches from *root*. For a search from *root* we now have to search the top index, and then each of the lower indexes:

$$\begin{aligned} F_{1u} &= T(K(1-(1-P)^{B^j-1})) + B^j T(K(1-(1-P)^{B^j-1})) + E(PN) \\ &= (B^j+1)T(K(1-(1-P)^{B^j-1})) + E(PN) \end{aligned}$$

Searches F_2 and F_3 are the same as in the replicated case.

The space required for each of the indexes at level *j* is the same, but the unreplicated top level index requires only space $S(B^j - 1)$.

$$\begin{aligned} S_u &= (B^j+1)S(B^j-1) \\ &= (B^j+1)K(c_s(1-(1-P)^{B^j-1}) + c_p P(B^j-1)) \end{aligned}$$

The Find operation for the single multiple-attribute index of Section 3 is a multi-step algorithm. The first step, finding the secondary index, is $T(K(1-(1-P)^N))$ time regardless of where we are in the database. The transitive closure of the reachability graph is inherently linear; for a search from root it will require time $O(B^j)$ from root, and constant time for the other searches.[1] Finding the appropriate objects in the secondary index can be done in two ways. If we are looking for objects reached from a large number of anchor points, a simple linear search may be desirable. If only looking for a few anchor points, we can use a typical index and perform a number of searches each of time $T(n)$, where n is the number of anchor points in the secondary index. Note that an anchor point will occur in a secondary index with probability $(1-(1-P)^d)$, where d is the number of objects directly beneath that anchor point. In our example, $d = B^j - 1$ for all the indexed locations, so $n = (B^j+1)(1-(1-P)^{B^j-1})$. Adding these up gives a find time of:

[1] We have described a technique where a tree-structured reachability graph can be replaced by ranges in a preorder numbering of the database. This would give constant, as opposed to linear, time and space requirements. Since this *only* applies to tree-structured data, we are not using this optimization for this analysis.

$$F_{1m} = T(K(1-(1-P)^N)) + t_r B^j + t_r (B^j+1)(1-(1-P)^{B^j-1}) + E(PN)$$

This is assuming that we make a linear search of the secondary index, otherwise the third term would change:

$$F_{1m} = T(K(1-(1-P)^N)) + t_r B^j + B^j T((B^j+1)(1-(1-P)^{B^j-1})) + E(PN)$$

Searches from children of root require the same time as the previous methods to get to the indexed locations, but beyond this we can make some optimizations. We only need to do the search in the primary index once. We will need to look at the reachability graph and perform a lookup in the secondary index once for each of the indexed nodes we reach. This gives a time of:

$$\begin{aligned} F_{2m} = {} & E(B^{j-1}-1) + T(K(1-(1-P)^N)) + B^{j-1}(t_r + T((B^j+1)(1-(1-P)^{B^j-1}))) + \\ & E(P(B^{2j-1}-1)) \end{aligned}$$

Searches from the bottom indexed locations also require the primary lookup, as well as a single check of the reachability graph and secondary index.

$$F_{3m} = T(K(1-(1-P)^N)) + t_r + T((B^j+1)(1-(1-P)^{B^j-1})) + E(P(B^j-1))$$

The space requirement here is a bit more complex. The reachability graph requires space proportional to the number of anchor points: $c_r(B^j+1)$. The primary index takes space for each search key, as well as a pointer to each secondary index: $(c_s+c_p)K(1-(1-P)^N)$. Each secondary index will take space determined by how many anchor points are found in the index and how many data items have the corresponding search key. The expected number of anchor points in an index is $(B^j+1)(1-(1-kprob)^{B^j-1})$, and the expected number of data items is NP. The space required for each item will be c_p, the cost of a pointer to a data item or anchor point. There will be one secondary index for each entry in the primary index. This gives a total space figure of:

$$\begin{aligned} S_m &= c_r(B^j+1) + (c_s+c_p)K(1-(1-P)^N) + \\ &\quad K(1-(1-P)^N)(c_p NP + (c_p NP + c_p(B^j+1)(1-(1-kprob)^{B^j-1}))) \\ &= c_r(B^j+1) + K(1-(1-P)^N)(c_s + c_p(1 + NP + (B^j+1)(1-(1-P)^{B^j-1}))) \end{aligned}$$

To understand the tradeoffs between the various indexing techniques it is helpful to graph the performance results on a particular scenario. There are many possible scenarios, corresponding to the values of the parameters on Pages 57 and 58. Given our space limitation, we will look at one representative scenario (a different scenario is presented in the experiments of the following section.) Therefore these graphs should be interpreted as illustrative only.

The graphs in the rest of 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. $T(n)$, the time for a lookup in an index, is logarithmic. $E(n)$, the time to search through n nodes in the database, is linear. We assume a main-memory database; with increasing memory sizes it is reasonable to cache “short” information, such as links and keywords, for each node in the database. Thus $E(n)$, the time to search through n nodes in the database, takes time $t_r \cdot n$. $T(n)$, the time to lookup a key in an index of size n, is logarithmic: $t_r \log_2(n)$. The factor t_r corresponds to memory lookup time, for these graphs we simply assume unit time.

Figure 6.4 shows the find time for each of the indexing methods, for a find over the entire database (F_1). We use $K = 1000$ and $P = .001$, this provides an expected value of 10 search keys per node.

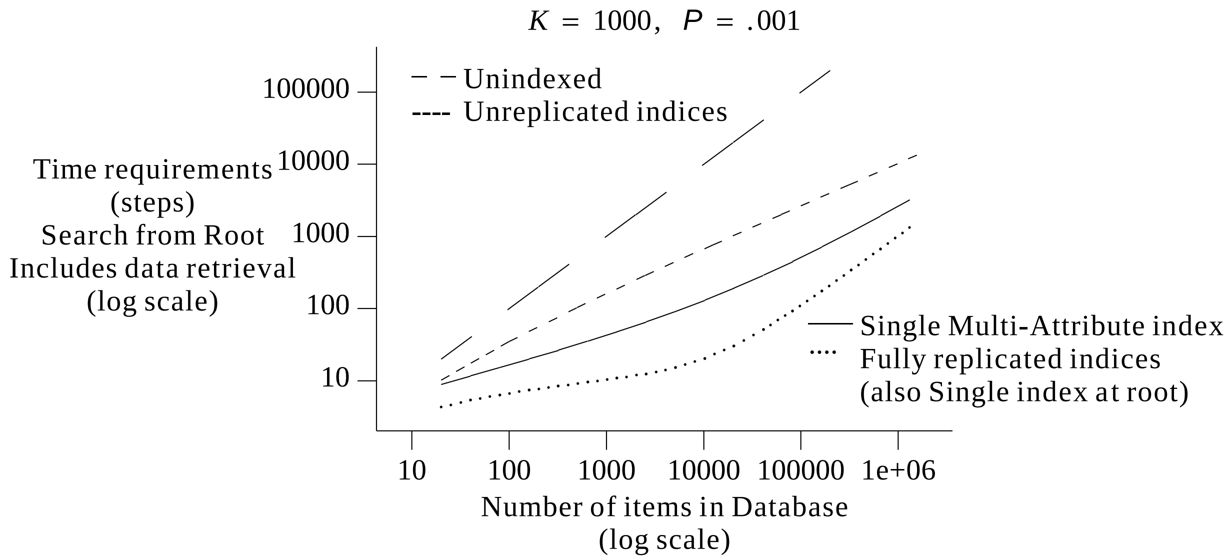


Figure 6.4: Find Time vs. Number of Data Items, search over entire database.

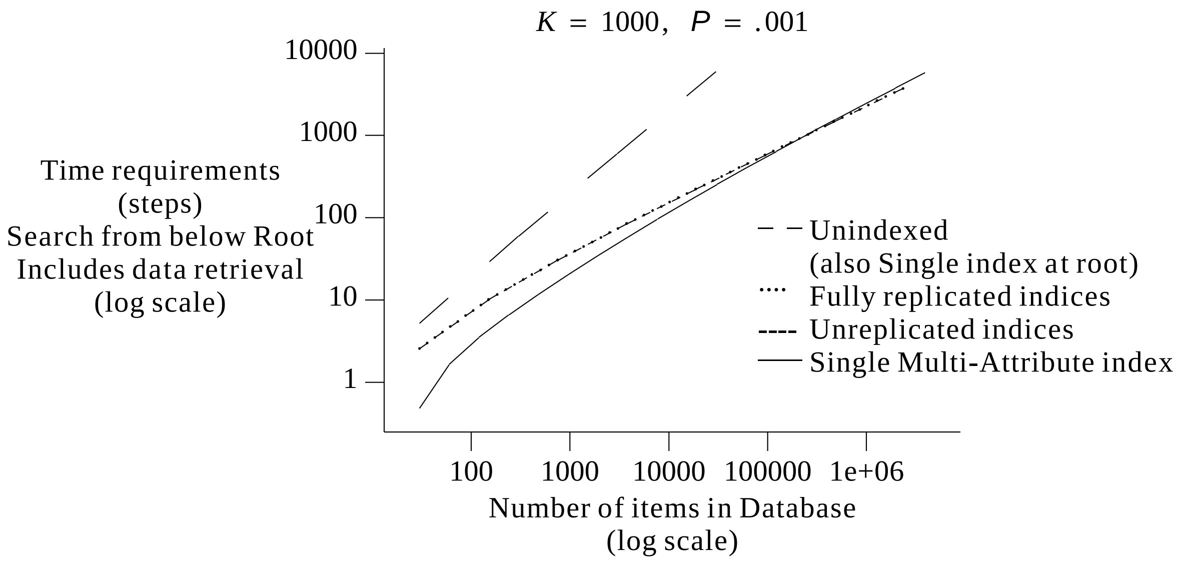


Figure 6.5: Find Time vs. Number of Data Items, search from just below root of database.

Figure 6.5 shows the expected time for queries from just below the root of the database (F_2, encompassing one fifth of the database.) Otherwise this figure corresponds exactly to Figure 6.4. The gains provided by indexing are substantial.

Figure 6.6 compares the effect of the number of distinct keys on the time required for a find. This is for the F_2 find, starting just below the root node. It does not include the actual object retrieval time, as this is the same for all of the indexes. The expected number of keys per node is constant (10); the more total keys, the fewer items will be returned for a given key. Note how the single multiple-attribute case performs better than the other methods with a large number of keys. Let us first explore what is happening with the fully replicated and unreplicated indexes. As the number of distinct keys grows, the size of each index grows. This increases the time required to search the indexes. This is also true with the multiple-attribute index, if we simply look at the search time for the *primary* index. However, the cost for the multiple-attribute method also includes a search based on the *secondary* index, and as the number of keys increases the size of each secondary index decreases. The cost of searching the secondary indexes decreases faster than the cost of searching the primary index increases. When the number of distinct keys approaches the size of the database, the cost of searching the secondary indexes becomes insignificant. At this point the cost of a single search in the (large) primary index of the

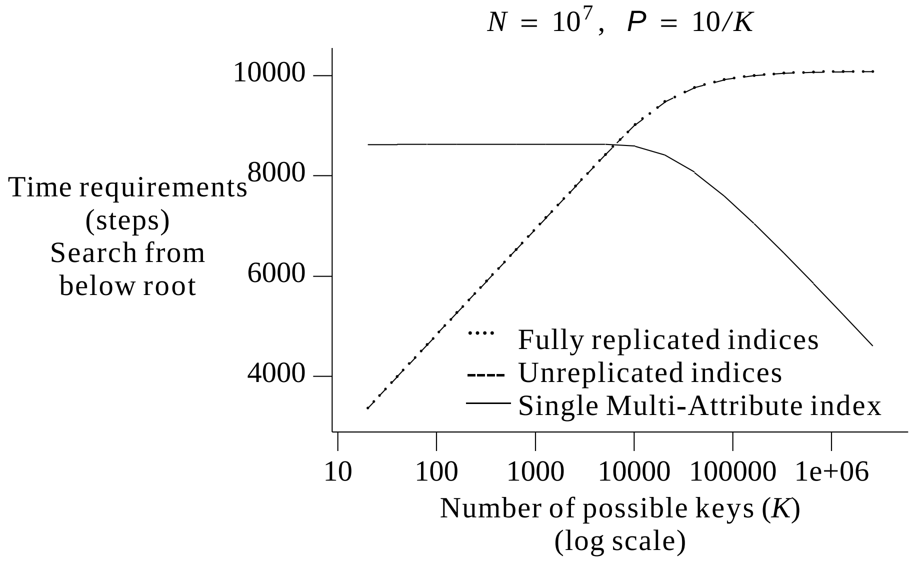


Figure 6.6: Find Time vs. Number of Keys, search from just below root of database.

single multiple-attribute technique becomes less than the cost of searching many lower-level indexes with the replicated and unreplicated methods.

The remaining figures show space requirements for the various methods. Figure 6.7 is space versus number of items in the database. We have assumed that $c_s = c_r = c_p = 1$

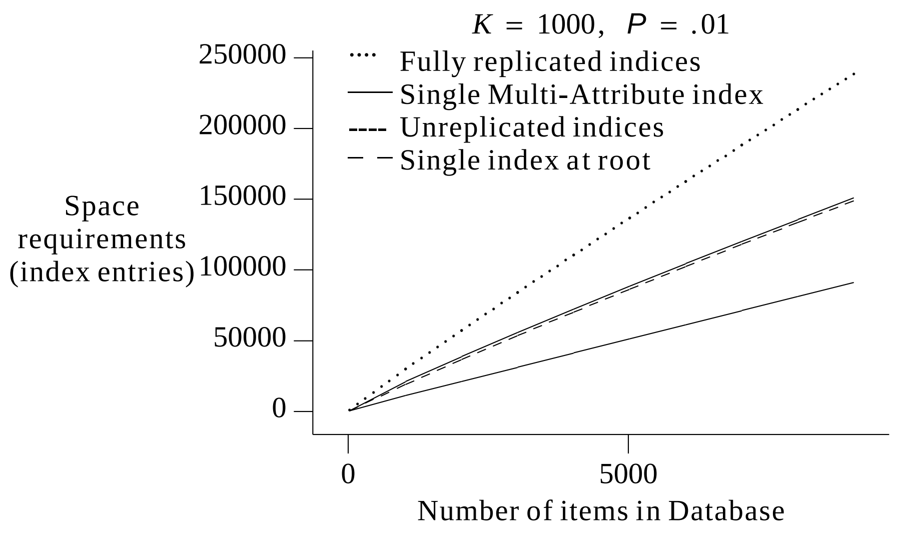


Figure 6.7: Index Space vs. Database Size.

word. For example, for a single index at root on a database of 5000 nodes takes 50,000 words, or about 10 words per object in the database. The database itself would take at least 75000 words, as each object would require a minimum of 15 words (10 keys and 5 links.) In practice a node will have much more information (such as text, other types of links, etc.), so the relative space cost of the index will be small.

Figure 6.8 corresponds to Figure 6.6, and shows space relative to the number of possible keys. This shows an interesting behavior; although the indexes grow as the number of distinct keys increases, the pattern of this growth is not obvious. If we look at the single index at root, we see that the space is relatively constant until the number of keys is comparable to the database size. Before this point, the size of the index is dominated by the storage of pointers to data items. Beyond this point, the cost of storing the keys dominates (as there are few data items per key.) With the unreplicated indices, the keys begin to dominate earlier, as each index covers a smaller area. Note that the curve for the fully replicated index is roughly the sum of the curves for the unreplicated indices and the single index at root. With the single multiple-attribute index, the space for the secondary indexes grows as well, resulting in the divergence between this method and the unreplicated indices.

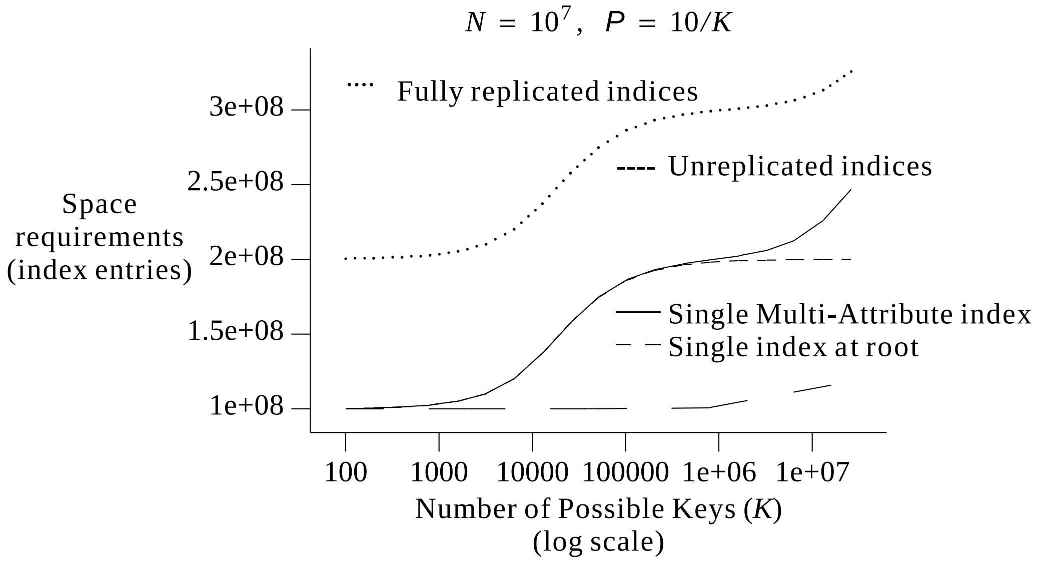


Figure 6.8: Index Space vs. Number of Keys.

From these graphs we can make a few interesting generalizations as to which index structure is best. The decision as to which index structure to use depends on the expected types of queries and how much storage space is available. The number and distribution of keys also has an effect on which method should be used. A single index uses the least space, but is only useful for F_1 type queries (unless searches use indexes above the start point, which was not considered by our analysis.) Replicated indexes provide the best or close to the best search times in most cases, at an expense in storage costs (about three times the space for a single index in our scenario.) The non-replicated indexes would be most useful when the majority of the searches start from low in the tree and space is at a premium. A multiple-attribute index strikes a balance between replicated and non-replicated indexes: It performs adequately on searches starting at *root* (F_1), but is slower for low starting queries (F_3). The space requirement is close to that of the non-replicated indexes.

6.1. Update Costs

Up to this point we have only discussed the cost of searching an index. Building and maintaining the index are very real costs, and cannot be ignored. The motivation for our work has come from databases that are dominated by reads, so we have concentrated on the search times. We do feel it is important to say something about the costs of building and updating indexes, however.

Building an index requires accessing every node reachable from the anchor point of that index. This is the same as the number of nodes accessed by a query using the index. If the cost of inserting an item into an index is not too large (logarithmic in the size of the index is a reasonable value), building an index will result in a net savings after running only a few queries.

Maintaining these indexes can be expensive. In some cases the cost of keeping an index coherent with a database update is as expensive as building the index. This depends on the type of update. The following paragraphs give time estimates, assuming that back pointers from data items to the index already exist (this cost was included in the space analysis above.) The costs are in terms of “number of index updates”. The time for a single index update varies with the type of indexing method; many of the methods in the literature may be used.

Adding or deleting a key from an item:

Replicated indices

This could require many updates: Every index that might reference the item must be modified, and an item is in the scope of every index on the path from root to that item.

Unreplicated indices

In this case, only one index points to any given data item, thus requiring only a single update.

Single multiple-attribute index

Here the object must be removed from or added to a secondary index, at a cost of a secondary index insert or delete, and a primary index find or insert (for insertion only.)

Adding or deleting a link:

Replicated indices

This could be expensive, as all indexes located above the changed node must be modified. If the change is small (such as adding or deleting a leaf), only an index insert or delete would be required. If a major portion of the graph is changed, however, the change could be as expensive as rebuilding each index from scratch.

Unreplicated indices

Here only a single index need be changed, but again the cost of that change varies.

Single multiple-attribute index

This requires modifying the reachability graph (a quick operation), and possibly modifying a number of secondary indexes. The number of secondary indexes to be modified would be the sum of all of the data items below the changed link, but above anchor points, plus all of the anchor points that are “first in line” beneath the changed link.

One factor to consider when judging the time “cost” of building and maintaining an index is the human factor. If an index is only used once, the cost to build it will outweigh the savings in terms of computer time; however the human cost of a delay in an interactive query may be substantial. Spending considerable off-hour batch time building indexes may be worthwhile even if the indexes are rarely used. Keeping an index coherent with updates can also be done off-peak; an index can simply be invalidated when an update occurs that might affect it.

6.2. Index Placement

So far in our cost analysis and experiments we have assumed a fixed index placement, with indexes at the root and halfway through the database. We tried experiments with randomly placed indexes, but performance was (not surprisingly) poor, as index "coverage" often overlapped and portions of the database were left unindexed. In a real database indexes would be placed at frequent search points, as determined by the user or Database Administrator. These points may not correspond to the index locations used in this analysis. Much of this analysis would still be relevant, but it is worthwhile to note one pitfall. With the non-replicated and multiple-attribute techniques, performance can suffer if too many indexes are used. In the non-replicated index case, this is because we have to search many small indexes. With the multiple-attribute method, the cost is in searching the reachability graph and secondary index. In practice this may not be a problem, as most searches may start from a few locations. Whoever (or whatever)[Fink88] is responsible for placement of the indexes must understand this in order to maximize the performance of the system.

It is possible to figure out how many indexes is "too many", in terms of actually decreasing performance. As an example, assume that we have one distinct key per item in the database, and a binary tree index (parameters $T(n) = \log_2(n) + c$, $E(n) = n$, $t_r = 1$, $K = N$, and $P = 1/N$.) Assume I index points with a roughly uniform placement (in the sense that each index directly covers the same number of items, which is true for the previous analysis.) This would give us an expected number of items that we have to *directly search* (before reaching anchor points in all directions) of $\frac{N}{2I}$.

With unreplicated indices, searching each index will take time $T(N/I)$. If we assume the query covers n total items, we can expect to search $\frac{n}{N}I$ indexes. This gives a total cost for the indexed search of:

$$\text{total search}(I) \ = \ E(\frac{N}{2 \cdot I}) + \frac{n}{N} I \cdot T(\frac{N}{I})$$

$$ts(I) = \frac{N}{2 \cdot I} + \frac{n}{N} I \cdot (log_2(\frac{N}{I}) + c)$$

We now need to find the value for *I* that minimizes the above function. We could do this by differentiating with respect to *I*, this gives:

$$\frac{d(ts(I))}{dI} = -\frac{N}{2I^2} + \frac{n}{N}(c + \log_2(\frac{N}{I}) - \log_2(e))$$

Finding the Roots of this function is a bit difficult. We can use numerical methods, however, to find minimums for $ts(I)$ for particular scenarios. For example, if we assume $N = 10^7$, and a query that covers roughly 10% of the data ($n = 10^6$), we find that performance drops off after about 2050 indexes.

With the multiple-attribute technique best we need not worry about the cost of searching the primary index (as it is done once regardless of the number if anchor points.) We have the same cost as above to reach all of the anchor points ($\frac{N}{2I}$), plus the cost of the reachability graph ($t_r \cdot \frac{n}{N} I$), and finally the lookup in the secondary index ($\frac{N}{2I}$, assuming a linear search.) This gives:

$$ts(I) = \frac{N}{I} + \frac{n \cdot I}{N}$$

We can find roots for this. Differentiating gives

$$\frac{d(ts(I))}{dI} = -\frac{N}{I^2} + \frac{n}{N}$$

This has a root at

$$I = \frac{N}{\sqrt{n}}$$

(The other root is with negative *I*, and is therefore uninteresting.) This technique encourages more indexes than with the non-replicated multiple indexes.

7. Experimental Results

The previous 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 experiments using our prototype query processor/main memory database on less regularly structured databases to verify this. The query processor was modified to detect queries that performed a transitive closure on a particular pointer type followed by a selection on a particular key; these queries then used any appropriate indexes. We include graphs in this section that plot the experimental results alongside predicted results from the analysis of the previous section.

The experiments presented here were run on a DEC 5410. As it is our goal to keep all search information in main memory, we wanted to run the experiments on a large memory machine to allow large databases (where indexing is most needed.) The IBM RTs used for the distributed experiments didn't have the memory we wanted for these experiments; the 128MB on the DEC allowed considerably larger tests.[2]

The experiments presented here serve two purposes:

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

In order to perform these experiments we must first calibrate the model, that is, determine the values for the time constants listed on Pages 57 and 58 that correspond to our prototype. 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 $E(n) = n \cdot 1.5 ms$. The index used for our experiments is a balanced binary search tree. We determined that the time to lookup an item in an index of size n is $T(n) = \log_2(n) \cdot 750 \mu u$.

In order to test our analysis relative to 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 that resembles a tree more than, for example, a completely connected graph. Therefore our experiments are based on data with a somewhat regular structure. We constructed two types of databases, trees and Directed

[2] It is worth noting that our prototype required no code changes to move to this new platform. We have also set up a distributed HyperFile database across a variety of hardware platforms, although we have run no controlled experiments in such a heterogeneous environment.

Acyclic Graphs. 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 3.)

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

Figure 6.9 gives results of F_1 queries (searches from *root*) performed on a tree-structured

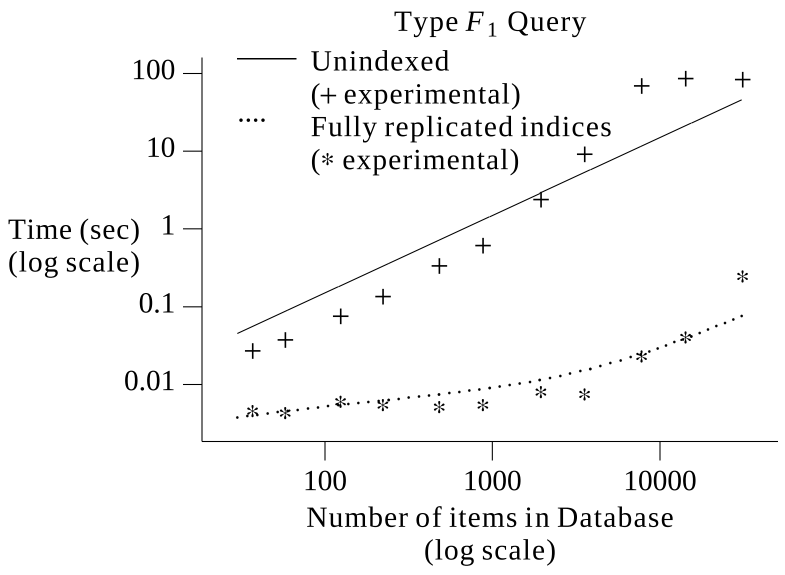


Figure 6.9: Queries from root on a tree-structured database.

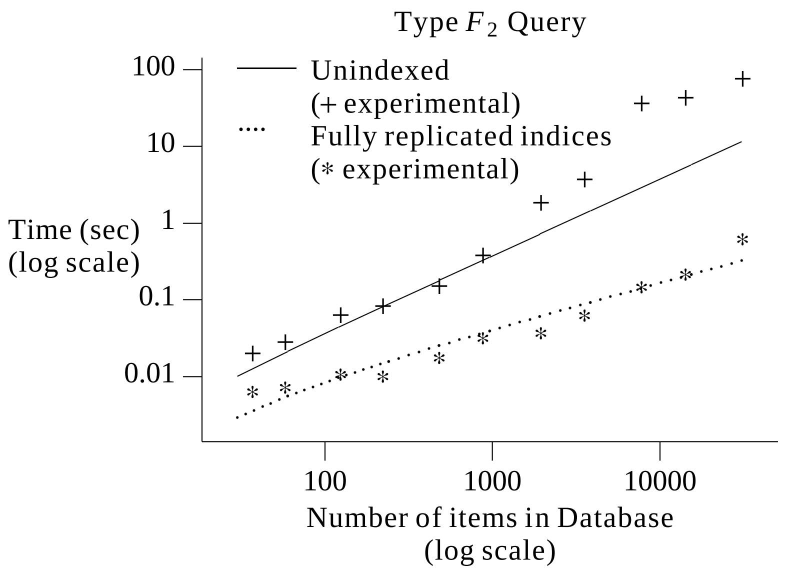


Figure 6.10: Queries from just below root on a tree-structured database.

database built to the above constraints. Figure 6.10 is for F_2 queries (searches from a node below the root) on the same data. The results for queries using indexes on small databases seem surprisingly low. Our best guess is that this is also partially a result of the machine architecture; we probably have a significant increase in the cache miss rate once the database exceeds a certain size.

We also tried queries on databases that were not tree-structured. To the databases used for Figures 6.9 and 6.10 we added links that form a directed acyclic graph rather than a tree. 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[Ande89]. 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.

Figures 6.11 and 6.12 show the results of queries run over these databases. The variation between the predicted and actual values is larger here than with the tree-structured database, however the predictions seem reasonable, particularly for 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.

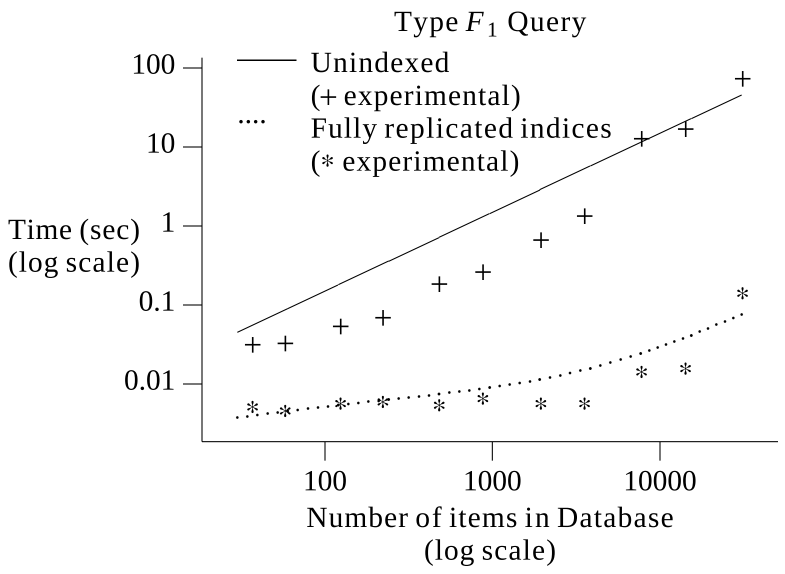


Figure 6.11: Queries from root on a DAG structured database.

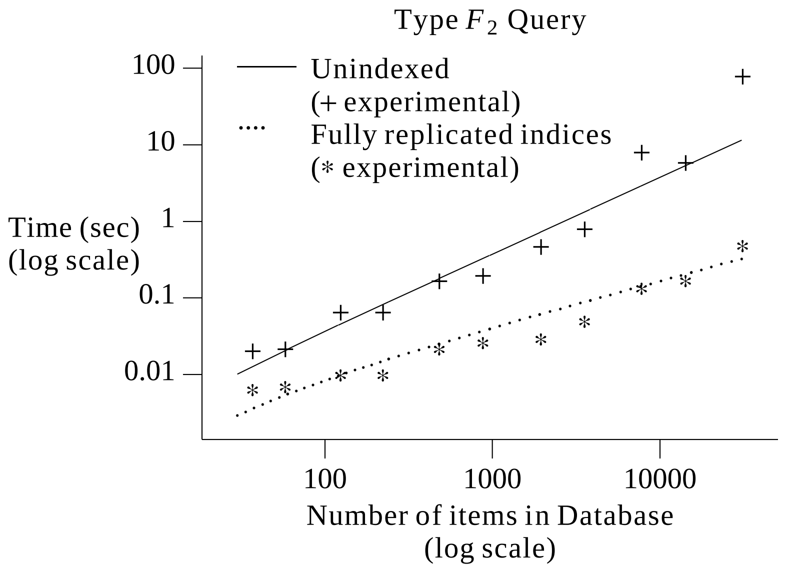


Figure 6.12: Queries from just below root on a DAG structured database.

The trends in the experiments coincide relatively well with the predictions from the analysis. The model we developed in Section 6 cannot be used to predict the exact performance of indexes on a particular database. Nevertheless, the model can be used to study tradeoffs and general trends.

8. Graph Structured Databases

The previous algorithms have been presented in the context of a tree-structured database. **D**irected **A**cyclic **G**raphs and arbitrary **D**irected **G**raphs present new problems. Figure 6.13 contains an example of the extensions we are talking about. Using only the solid lines gives us the familiar tree structure. Adding the dashed links gives a DAG, and adding the dotted links gives a DG. Index creation is relatively easy, as we need only mark items as being in the index when they are first inserted. Updates to the primary attribute are unchanged, but link deletion becomes more difficult. This is not a serious problem as it is related to Garbage Collection, which has been studied extensively[Cohe81]. We will briefly summarize some possible solutions.

8.1. Directed Acyclic Graphs

With DAGs, we can attach a reference count to the back-pointer from an object to an index noting how many ways it is *directly* reached from that index. The reference count of an object is the number of parents it has that are in the index, regardless of the reference counts of the parents. When an item is told by its parent that the parent is no longer in the index (or the link between the parent and child is broken), it decrements its reference count. Only when the count reaches 0 is the object deleted.

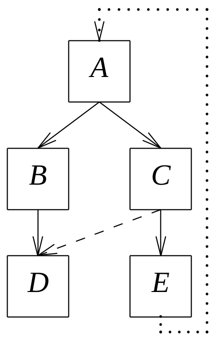


Figure 6.13: Arbitrary directed graph database.

8.2. Directed Graphs

The problem here is with cycles. A simple solution to deletion in this case is to re-create the index any time a link is deleted. This is only necessary when the deleted link may have been part of a cycle. In other cases, the reference count mentioned for the DAG case is sufficient. Cycles, and the deletion of links therein, are probably infrequent enough that this will be adequate in practice.

CHAPTER 7

Other Issues

We have discussed the major issues in designing HyperFile. In this Chapter we touch on a number of minor issues, such as version management and implementation concerns.

1. Versions

Keeping track of updates to an object is an interesting problem. In many situations it is desirable to maintain a version history. We will outline a method that allows versions and historical/version queries to be represented within the HyperFile model presented in Chapter 2.

An object *O* becomes "versioned" by adding a tuple (version, *<timestamp>, <pointer to previous version>*) to the object. Initially, the previous version pointer is null, and the timestamp is the creation time of the object. When an update to is made to *O*, the application first looks for a version tuple. If one exists, a copy *O′* is made of object *O*. The original *O* is updated in place, and its version tuple is modified: The timestamp gets the current time and the pointer points to the old version *O′*. Note that pointers to *O* always point to the latest version. An example is shown in Figure 7.1.

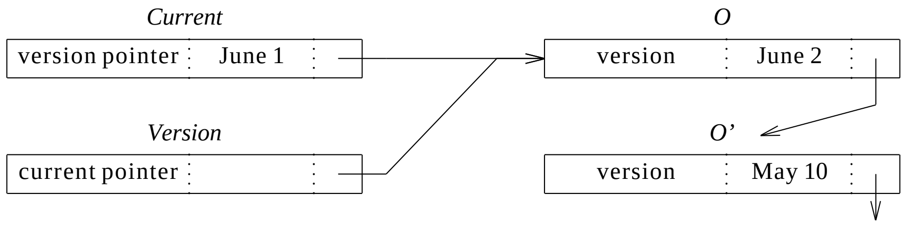


Figure 7.1: Versioned object.

We can use this to provide two types of pointers to versioned objects: Those that point to the *latest* version (the standard ones we have used so far), and those that point to a *specific* version of an object. For example, the working version of a paper would have pointers to the latest version of each section. A submitted paper would be “frozen”, the pointers would be fixed to the current version of each section. Updates could continue on the working draft, however.

Pointers to a *specific* version are created using a tuple (version pointer, *<timestamp>, <pointer>*). Such a tuple is dereferenced with a query that first goes to the latest version (pointed to by *<pointer>*), then checks the **version** tuple in the object to see if its timestamp is before the timestamp of the version pointer. If not, the pointer in the version tuple is followed. This continues until the proper version of the object is found. To be more precise, the algorithm to dereference a version pointer tuple is as follows:

```
Dereference( version pointer, Vp_timestamp, Vp ):
    Select ("version", time, previous) from object Vp
    If tuple does not exist or                  ;; Not a versioned object
       $time \leq Vp\_timestamp$ then            ;; Proper version
           return Vp
       else
           return Dereference( version pointer, Vp_timestamp, previous )
```

Note that this does not require a change to the semantics of the language, simply a slightly more extensive query. For example, the basic query to find all items pointed to by items in the set S is:

$$S \mid (\text{version pointer}, ?, ?x) \mid \uparrow x \rightarrow T$$

Note that this assumes we are only following version pointers. This will give the latest version, regardless of the timestamp of the version pointer. However, we can also issue a query that will in fact give the desired version:

$$S \mid (\text{version pointer}, ?ts, ?x)\ [\ \mid \uparrow\uparrow x \mid (\text{version}, ? > ts, ?x)\ \text{OR}\ (\text{version}, ?, ?)\ ]^{*}$$
$$\mid (\text{version}, ? \leq ts, ?) \rightarrow T$$

This first sets and dereferences the version pointer. The dereferenced object is checked to see if it is *newer* than the desired one, if so the pointer is dereferenced (the OR clause allows the correct version to slip through **without** setting a pointer for dereferencing.) Finally, all versions newer than the desired one are thrown out, leaving the correct version.

Even though the fundamental model supports versions with no change, it may be useful for an implementation to provide special support for versions. Requiring pointers to old versions to go through the timestamp dereferencing process, as opposed to pointing directly to the old version, allows old versions to be stored as "Δ's" (differences from the current version.) The *complete* old versions can be built as part of the dereferencing process.

Queries can also choose specific versions manually; for example

$$S\ [\ |\ (\text{version},\ ?\ \geq 10{:}00,\ ?x)\ |\ \uparrow\uparrow x\ ]^{*} \rightarrow T$$

chooses all versions of objects in S that were in effect on or after 10:00.

Versions can also be used for "historical queries". This is done by giving version tuples in the query an explicit timestamp (for example, "February 3, 1959") instead of using the timestamp of the version pointer. In some cases this will cause the final null pointer to be followed, and no object to be returned. This makes sense, as in this case the item was created *after* the time of the query.

With many simultaneous users of a large database, some form of concurrency control mechanism is needed. Lock based transactions may be inappropriate in many cases. Long "editing" updates would result in long-running transactions that could tie up portions of the database for an unreasonable time. This may be unnecessary, particularly with modifications that do not change "critical" data, such as changing the wording of a document. Transaction management schemes that are more flexible than serializability have been proposed[Kort88, Horn87], many of these may be applicable to HyperFile. We can also use versions to address these problems.

Suppose a new version of a document appears while a query is running. The old version may have been accessed as part of the query, and another pointer to the object followed after the new version is in place. Should the new version be ignored, should the old version be ignored, or should both be included? The last option is easy, just send out whatever result is obtained as soon as it has been determined to be part of the result. This forces the application to deal with multiple versions of some objects. Using only new versions requires saving results until query termination. This eliminates one of the advantages of HyperFile queries: allowing an application to begin displaying results before a query is complete. If a new version of a processed document is installed before the query terminates (determined by noting if the old version has been *marked* by the query) the old version would have to be discarded and the new version processed. A greater difficulty is with pointers that have been followed from the old version, but are not present in the new version. These would have to be “unqueried”, leading to a cascading partial-abort of the query. This requires maintaining considerable state beyond a simple mark of the document. The mark would have to note which objects had referenced the marked object, so as to determine if a partial-abort is applicable, or if the object would have been processed through another part of the query anyway. As one of the advantages of this system is the simplicity of query processing, we see this as a poor option.

Historical queries can solve this dilemma. *All* queries (or at least those that need consistency) are run as historical queries. If a particular time is not given, the current time is assigned as a query timestamp. This gives a measure of consistency; the results of a query are as if it were executed atomically at the time of the timestamp (with respect to versioned objects.)

In a system without globally synchronized clocks, some measure of consistency can be obtained by placing a local timestamp on each query when it first arrives at a site, and using only versions active at that time. This ensures consistency between objects at a single site, requires no extra message traffic for clock synchronization, and does not require any special guarantees (such as ordered message delivery) from the communications system.

2. Large Memories

This research was done as part of the Massive Memory Machine project at Princeton[Garc84]. The basis of this project is that memory is becoming less expensive; having a gigabyte of main memory on a workstation is not unreasonable in the near future. In particular, available main memory is likely to increase at a significantly faster rate than processor speed, disk size, or disk access rate.

In the case of HyperFile, the availability of large memories enables us to cache most or all of the “search” information (keywords, links, etc.) in main memory. As a result disk reads will only occur when an application needs specific large pieces of data, which will generally be after the desired objects have been found (as opposed to a file system, where searching through the entire text of many files is often necessary if you don’t remember the name of the file you want.)

As an example, the Princeton University Library has over 4 million books[PUGS90]. Assuming a typical book has 100 pages, and about 3000 characters per page, storing all of this data on-line requires roughly a terabyte. Compression techniques could cut this considerably, however a book full of pictures will greatly expand this amount. In any case, it is safe to assume that we are a long ways from being able to store a significant library in main memory, and an interactive search through a terabyte of disk is not something any of us wish to contemplate.

The key search information (Title, authors, references, keywords, etc.) for a single book can probably be stored in less than 1k bytes, however. This allows our gigabyte workstation to store ¼ of the Princeton University library; four such workstations (or a single 4 gigabyte machine) could handle all of the searching on-line. Note that this goes way beyond the search capability provided by a typical on-line card catalog, not only are more complex queries allowed, but the desired document is available on-line once found.

Obviously, such a system is not likely in the near future, due to social, legal, and practical issues. Nevertheless, we can see that the technological barriers to such a system are falling.

3. Data Types

The data model provides for triples composed of items from a few basic types. HyperFile has a limited basic type system. This is less general than, say, an object oriented system. Unstructured types (similar to files in a files system) are provided, however. This gives a high degree of flexibility -- it is the *constraints* that are limited by the “limited” type system, not the data that may be put into HyperFile. This gives advantages in efficiency (the server is less complex), as well as advantages in flexibility (applications do not have to fit a predefined database schema.)

All of the operations on these types are not defined in this thesis. The appropriate operations are obvious in most cases, and this work presents no novel approaches other than those mentioned here.

3.1. Key field types

The following data types can be operated on directly in non-trivial ways by HyperFile. They are intended for use in the **key** field in a triple.

3.1.1. Word

Words can be thought of as short character strings. Typically these will be used for keywords or index entries. Standard string operations will be allowed on these. In particular, expression matching will be a common operation.

Words are not arbitrary character strings, however. By placing some restrictions on the scope of words we can represent them more compactly and search them more efficiently than a character string representation would allow.

To do this we make use of PATRICIA[Morr68] (a search tree algorithm) to build a *two-way hash function* that allows us to map a limited set of strings to a fixed-size representation and back. We will not describe this algorithm in detail, other than to note that it allows us to efficiently ($O(\textit{string length})$) map strings of characters into integers and vice-versa, preserving order. This allows our Word type to support some pattern matching (such as *ca** to match *cat* and *cab*, or *'cab < ? < cat'* to find *cad*.) A trie for the above example is given in Figure 7.2; note how *cab* maps to 30, *cad* maps to 40, and *cat* maps to 50.

Carefully setting the mapping for the initial tree allows considerable flexibility in adding new words as time goes along, for example in Figure 7.2 we have room for 9 new words between each of *cab*, *cad*, and *cat*. Given a storage size of 32 bits, we can have potentially 2^{32} Words. In practice we will not do this well, but a typical HyperFile database will probably not contain nearly so many distinct keywords. Credit must go to the on-line Oxford English Dictionary project at the University of Waterloo for providing an example of the use of PATRICIA in the context of document database, inspiring our ideas[Raym88].

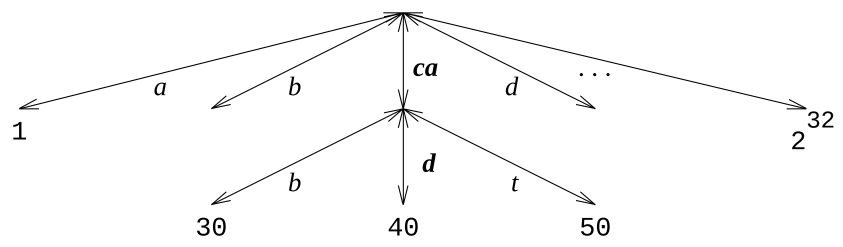


Figure 7.2: Trie for mapping Words to integers.

3.1.2. Numeric

Numeric data and standard numeric operations are supported. Currently we support a single numeric type, although dividing this into separate real and integer types would be straightforward. We see little use for real data as a structured part of HyperFile objects at this time.

Other types, such as dates, can be implemented on top of numbers in a straightforward manner. This allows some application-driven “extensibility” of the type system.

3.1.3. Pointer

*Pointer*s are perhaps the primary thing that sets HyperFile apart from most data servers. Note that since a pointer is actually embedded in a triple, the type and key (or type and data) fields can be used to attach information to the link.

> *One method of specifying that a document contains another.*
>
> (pointer, "contains", <*pointer to sub-document*>)
>
> *More complex method, using triple-type to specify information.*
>
> (chapter, "One", <*pointer to chapter (sub-document)*>)
> (Appendix, "A", <*pointer to appendix*>)

3.2. Data field types

The data field of a tuple can contain any of the above types, although the query processing engine is optimized for searches on the key field. In addition, the data field can contain larger data items. Although of variable length, most are relatively compact, the exception being *text*. In our prototype, everything except text data items is cached in memory. This speeds queries that do not require looking at text items.

3.2.1. String

Strings are intended to be short sequences of characters; at most a sentence. These will be commonly used for titles, names, addresses, and other such data. Standard expression matching and string operations will be allowed, but operations can be expected to be *much* slower than equivalent operations on **Word**s.

3.2.2. Short

This is a short unstructured type, where the data is of a small (although not necessarily fixed) size. The idea is that this can be used to support extensible data types. Searches will be allowed on this data by allowing the application to provide functions to determine matches.

There are a number of possibilities for when to determine the size of *short* blocks. Possibilities are:

- Fixed for all implementations.
- Fixed for a given site.
- Fixed for a given application.
- Fixed at the creation of the triple.
- Automatically varying.
- Application-specified, but variable (the size would be specified when the triple is created, and an operation would be provided to “grow” and “shrink” the object.)

We prefer the last option; unstructured data items are flexible, but the application writer is made aware of the expense of changing the size of the data. Having a fixed size would be easier to implement, but would limit the application, and may force objects into the *text* type that don’t belong there.

Since the above types will occasionally be used for searching (although not as frequently as items in the **key** field), they should be cached in memory along with the rest of the document. However, since they are not fixed length (and the rest of the data items are) they will be kept separately.

3.2.3. Text

This is the basic unstructured type. Any operations that must be performed on text blocks (other than creation and deletion) must be provided by the application. Text can be viewed simply as a string of bits. Although initially intended for use as a medium for the written word, text blocks can be used for pictures, executable code, or other data that does not fit into the normal type system.

All data other than text will be cached in memory when possible. Thus searches on text data will be comparatively slow. HyperFile does not operate directly on text, it only provides it to the application. This encourages the application writer to avoid using the text data type for data that is used for searching.

In our prototype, text is stored as a file. This allows the use of existing tools when building applications such as editors and text formatters, as well as simplifying prototyping. This is not simply a prototyping decision, however. Providing a file interface allows applications to be converted for use with HyperFile by simply wrapping them in a shell that handles the query interface, and redirects file system calls to the appropriate HyperFile text block.

4. Triple types

These are the actual key-data combination types, each designed with a particular purpose in mind (although not limited to the original purpose.) The type identifies the **meaning** of the triple; multiple triple types may exist that have the same underlying physical types for the key and data.

Individual applications define triple types as appropriate. Types extend across the entire database, however, and thus conflicts must be resolved between applications that wish to use the same type name to mean different things. This conflict resolution also serves to encourage the sharing of data between applications. The definition of triple types is in some sense the “schema” of a HyperFile database.

In order to ease the problem of type definition, each database has a “reserved” catalog document. This contains the actual specification of each triple type, including the type name and the physical types of the key and data. In addition, since the catalog is a document, a written explanation of each type is provided. Although this does not automatically resolve conflicts in the use of triple types, it does simplify human resolution of the problems, as a textual (or even multimedia) definition of the type is contained in the database. Although I hesitate to say that the schema is self-documenting, this method does encourages whoever defines a type to document it, as the catalog must be updated in order to make the type usable at all.

Certain types will be predefined. In particular, descriptions of the types used in the catalog will be always be contained in the catalog. A sample catalog is contained in Figure 7.3. A relational model would probably be better for expressing the information in the catalog, but it should be remembered that the catalog has a primary purpose other than as a document. It is a statement of the power of the model that this information can be expressed at all.

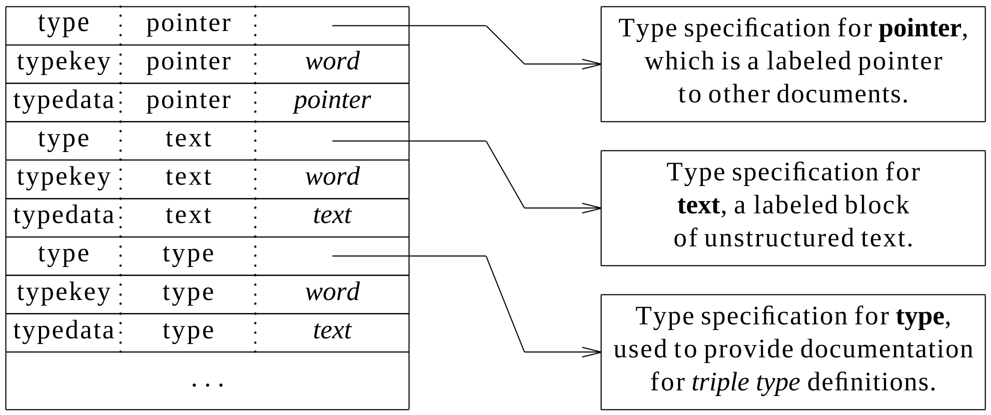

type : pointer :
typekey : pointer : *word*
typedata : pointer : *pointer*
type : text :
typekey : text : *word*
typedata : text : *text*
type : type :
typekey : type : *word*
typedata : type : *text*
. . .
Type specification for **pointer**, which is a labeled pointer to other documents.
Type specification for **text**, a labeled block of unstructured text.
Type specification for **type**, used to provide documentation for *triple type* definitions.

Figure 7.3: Sample Catalog.

CHAPTER 8

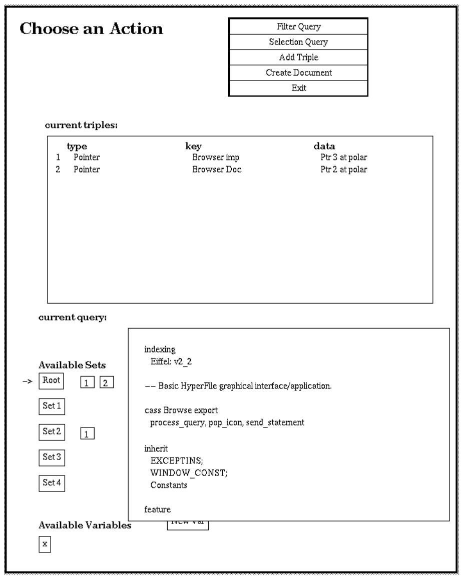
A Browsing Application for HyperFile[†]

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; 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 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 8.1 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

[†] The application presented in this Chapter was implemented by David Bloom ’91 as a Junior Project at Princeton under the supervision of Hector Garcia-Molina and the Author. It is included here for completeness in describing HyperFile, but should be viewed as collaborative work rather than original work of the Author.



	type	key	data
1	Pointer	Browser imp	Ptr 3 at polar
2	Pointer	Browser Doc	Ptr 2 at polar

```
indexing
  Eiffel: v2_2

-- Basic HyperFile graphical interface/application.

cass Browse export
  process_query, pop_icon, send_statement

inherit
  EXCEPTINS;
  WINDOW_CONST;
  Constants

feature
```

Figure 8.1: Complete browser screen.

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

Root [| (Pointer, 'Calls', ?X) | $\uparrow\uparrow$X]2 $\rightarrow$ Z

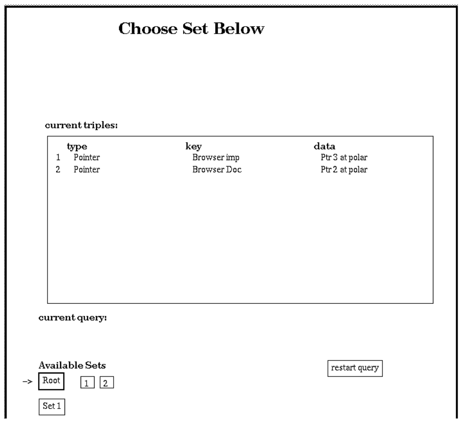
We will then look through these routines for those written by David, and take a look at the code of one of those routines with the following query:

Z | (String, 'Author', "David Bloom") | (Sources, 'Eiffel', $\rightarrow$code)
    { display_text (code) }

Instead of typing queries, the browser lets us enter queries interactively using menus. The menu at the top of Figure 8.1 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 Chapter):



Choose Set Below

current triples:

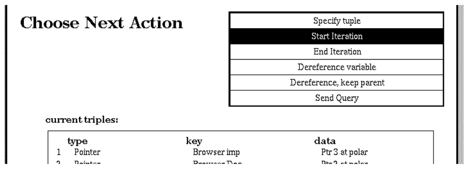
	type	key	data
1	Pointer	Browser imp	Ptr 3 at polar
2	Pointer	Browser Doc	Ptr 2 at polar

current query:

Available Sets

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

We now choose the criteria we wish to select on:



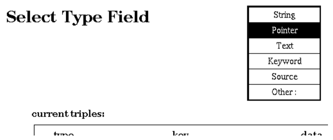
Choose Next Action

current triples:

	type	key	data
1	Pointer	Browser imp	Ptr 3 at polar

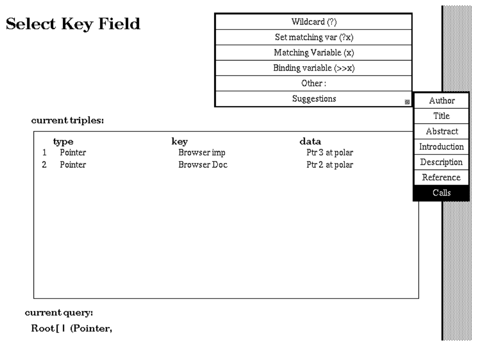
In this case, we will start by iterating (since we will want to follow pointers recursively.) We are then returned to the same menu, and choose **Specify Tuple**. This allows us to specify criteria that restrict the objects we are interested in; objects that do not meet this criteria will be ignored. In this case, we want to follow **Pointers** to **Call**ed routines.

We are immediately prompted for the *type* of the tuple we wish to search on:



Note that the 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*.



current triples:

	type	key	data
1	Pointer	Browser imp	Ptr 3 at polar
2	Pointer	Browser Doc	Ptr 2 at polar

current query:

Root [| (Pointer,

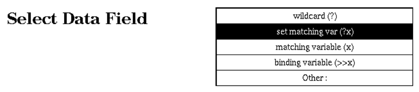
Note that we have a number of options:

Wildcard:	Accept any key.
Set Matching Var:	Set a variable that 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 that 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 will just pick **Calls** from the suggestions menu.

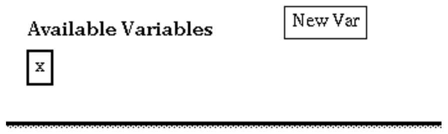
It is also worth noting that the partially completed query is displayed as it is constructed; this is shown at the bottom of the preceding figure (under **current query:**.)

We next have to specify the data field. In most cases, this is a long field, such as the text of a paper. As a result, comparisons will be infrequent. In the case of a *Pointer* tuple, however, 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:



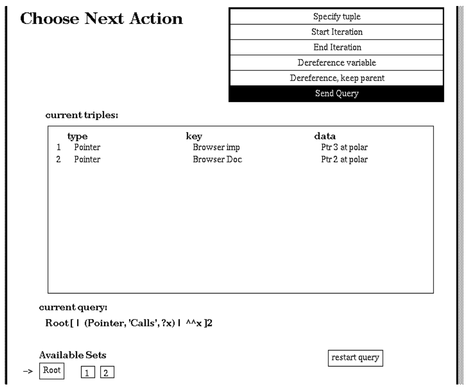
An unused identifier is assigned for this variable, and inserted in the **Available Variables** area (shown at the bottom of Figure 8.1.)

We have now completely specified the tuple for this filter. This returns us to the **Next Action** menu. We have now picked out the pointers we want to follow, so we choose **Dereference, keep parent**. This will add 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 (no sense gathering the entire code just for an example.) We are then ready to send the finished query:



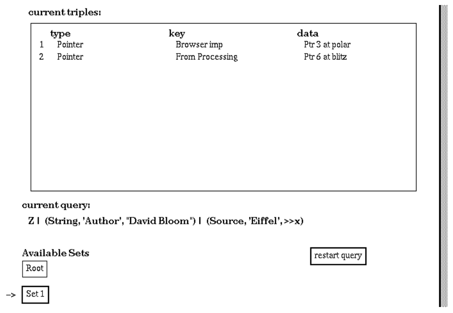
current triples:

	type	key	data
1	Pointer	Browser imp	Ptr 3 at polar
2	Pointer	Browser Doc	Ptr 2 at polar

current query:

```
Root [ | (Pointer, 'Calls', ?x) | ^^x ]2
```

The results of this query are displayed in the window at the top of Figure 8.2. 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



current triples:

	type	key	data
1	Pointer	Browser imp	Ptr 3 at polar
2	Pointer	From Processing	Ptr 6 at blitz

current query:

```
Z | (String, 'Author', "David Bloom") | (Source, 'Eiffel', >>x)
```

Figure 8.2: Result of recursive query.

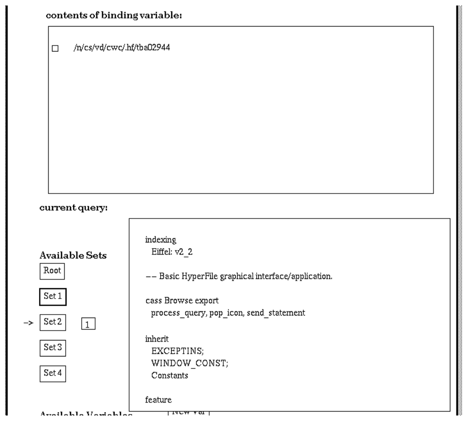
future use. Currently "next available" is the least recently used set; other options (such as allowing the user to specify which set) could be used. An arrow points to Set1, to show that it is the currently displayed set (as shown in the bottom of Figure 8.2.)

Figure 8.2 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) that 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 is retrieved explicitly (*(Source, 'Eiffel', →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 8.3.) 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. Therefore Set2 would be useful as a set (with a single object) for the



/n/cs/vd/cwc/.hf/tba02944

```
indexing
  Eiffel: v2_2

-- Basic HyperFile graphical interface/application.

cass Browse export
  process_query, pop_icon, send_statement

inherit
  EXCEPTINS;
  WINDOW_CONST;
  Constants

feature
```

Figure 8.3: *Viewing a binding variable.*

start of another filter query.

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 that 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 ideas of menu-based query construction and *hints* serves as a good basis for forming application-specific queries.

CHAPTER 9

Conclusions

We have described HyperFile, a back-end data service for heterogeneous applications. It provides a query language that permits searches based on properties of the stored objects, as well as by following pointers contained in the objects. We believe that the query language is powerful enough so that many common queries in applications such as document processing can be answered with a single request to HyperFile. Yet, HyperFile is simple and flexible enough that designers of such applications will not have to resort to file systems to store their data.

The query processing algorithm we have presented is straightforward and efficient. The language was intentionally limited in order to allow fast query processing. We believe this is a good approach for a data *server*. Nevertheless, there is room for more work on HyperFile query processing. We have not looked at query optimization; this could be an interesting area. There is even more room for work on optimizing distributed query processing. As an example, our technique requires keeping little information about non-local objects, but multiple references to the same remote object will result in unneeded messages. Sharing information may eliminate some of these, but this would add complexity. Another potential optimization is saving remote references until local processing is complete, rather than sending queries immediately. This would save cut communications cost, but could also decrease parallelism. This brings up another issue: HyperFile queries should be able to achieve considerable parallelism on a multiprocessor. Adapting our query processing algorithm to different multiprocessor architectures could be interesting work.

We have described techniques for indexing HyperFile queries. These techniques are not necessarily optimal, efficient means of precomputing transitive closure would help considerably. There are significant limits to what can be done for general graph structures, but improvement is possible for some types of graphs. We may be able to use this to generate techniques for transitive closure that have good performance in typical database situations. Another area that we have not fully explored is limited depth indexing (as opposed to complete transitive closure.) In fact our HyperFile implementation is quite limited in what types of queries it will use indexes for. Expanding this a problem related to query optimization. We can also explore other applications for these indexing techniques.

We have presented a sample user interface to generate HyperFile queries. In practice the types of queries, and how they are generated, will be application specific. Further information on the utility of HyperFile can be gained by using it to support specific applications. We are currently looking at using HyperFile to support scientific data. Scientific environments are often heterogeneous in hardware, software, and perhaps most important, personnel. Although such environments have made use of traditional business-oriented databases, rarely is 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.

There are many data management applications that are not well served by traditional database management systems. A few examples are software engineering (code, design data, executables); computer aided design (geometric data, test information); and animation (linked graphics frames, evolving video compression techniques.) Special purpose database technologies, such as spatial database systems, are being developed to support these applications. Integrating these new technologies into a single database is difficult. HyperFile provides a means for integrating these types of data, by serving as an underlying storage service that can track the relationships between the widely varied application specific data.

There will be data management areas for which HyperFile is not appropriate. Nevertheless, we have learned one thing that will be useful in developing data managers for these areas. Existing database technology can be reused and applied in new ways. For example, the indexing methods we have presented can make use of existing research in both indexing and transitive closure. Distributing HyperFile was also simplified by using existing technology, for example our naming scheme. We may never have a “perfect” database management system that adequately supports all applications, but by combining existing technology with new ideas the community of database researchers should be able to keep pace with the data management needs of new applications of computers.

APPENDIX A

BNF description of HyperFile Interface Language

statement ::= expr $\rightarrow$ *object*
          ::= expr

expr ::= *object*
     ::= expr setop expr

     ::= expr filterexp
     ::= *object* selector

setop ::= $\cup$, $\cap$, $-$

filterexp ::= filterexp filterexp
          ::= **[** filterexp **]**^{n}
          ::= **[** filterexp **]***
          ::= **|** filter

filter ::= **(**typespec**,** key**,** data**)**
       ::= filter **or** filter
       ::= **not** filter
       ::= arrow

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

arrow ::= $\uparrow$filtervar
      ::= $\uparrow\uparrow$filtervar

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

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

typespec ::= *name of type of this triple*
         ::= application-communication

application-communication ::= $\rightarrow$*identifier* *Send value to application.*

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

filtervar ::= *identifier*

Bibliography

DBTG74. Data Base Task Group, "CODASYL Data Description Language," NBS Handbook 113, National Bureau of Standards, US Department of Commerce, Washington, DC (January 1974).

PUGS90. *Graduate School Announcement*, Princeton University, Princeton, NJ(June 2, 1990), p. 23.

Aksc88. Robert M. Akscyn, Donald L. McCracken, and Elise A. Yoder, "KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations," *Communications of the ACM* **31**(7)(July 1988).

Allm76. Eric Allman, Michael Stonebraker, and Gerald Held, "Embedding a Relational Data Sublanguage in a General Purpose Programming Language," pp. 25-35 in *Proceedings of the Conference on Data: Abstraction, Definition, and Structure*, ACM (March 22-24, 1976). Also SIGPLAN Notices **8**(2):II.

Ande89. T. Lougenia Anderson, Arne J. Berre, Moira Mallison, Harry Porter, and Bruce Schneider, "The Tektronix HyperModel Benchmark Specification," Technical Report No. 89-05, Tektronix Computer Research Laboratory, Beaverton, OR (August 3, 1989).

Birr82. Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder, "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM* **254**(4) pp. 260-274 (April 1982).

Carr86. Nicholas Carriero and David Gelernter, "The S/Net's Linda Kernel," *Transactions on Computer Systems* **4**(2) pp. 110-129 ACM, (May 1986).

Clif88. Chris Clifton, Hector Garcia-Molina, and Robert Hagmann, "The Design of a Document Database," pp. 125-134 in *Proceedings of the Conference on Document Processing Systems*, ACM, Santa Fe, New Mexico (December 5-9, 1988).

Clif90. Chris Clifton and Hector Garcia-Molina, "Indexing in a Hypertext Database," pp. 36-49 in *Proceedings of the 1990 International Conference on Very Large Databases*, VLDB, Brisbane, Australia (August 13-16 1990).

Clif91. Chris Clifton and Hector Garcia-Molina, "Distributed Processing of Filtering Queries in HyperFile," in *Proceedings of the International Conference on Distributed Computing Systems*, IEEE, Arlington, Texas (May 20-24, 1991).

Codd70. E. F. Codd, "A Relational Model for Large Shared Data Banks," *Communications of the ACM* **13**(6) pp. 377-387 (June 1970).

Cohe81. Jacques Cohen, "Garbage Collection of Linked Data Structures," *Computing Surveys* **13**(3) pp. 341-367 ACM, (September 1981).

Crof87. W. B. Croft and D. D. Lewis, "An Approach to Natural Language Processing for Document Retrieval," pp. 26-32 in *Proceedings of the 10th Annual*

International ACM SIGIR Conference on Research and Development in Information Retrieval, , New Orleans, LA (June 1987).

Cruz87. Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood, "A Graphical Query Language Supporting Recursion," pp. 323-330 in *Proceedings of the SIGMOD International Conference on Management of Data*, ACM, San Francisco, CA (May 27-29, 1987).

Dada86. P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walch, "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables an Hierarchies," pp. 356-364 in *Proceedings of the SIGMOD International Conference on Management of Data*, ACM, Washington, DC (May 28-30, 1986).

Elli77. C. A. Ellis, "Consistency and Correctness of Duplicate Database Systems," *6th Symposium on Operating System Principles*, pp. 67-84 (1977).

Fink88. S. Finkelstein, M. Schkolnick, and P. Tiberio, "Physical Database Design for Relational Databases," *Transactions on Database Systems* **13**(1) pp. 91-128 ACM, (March 1988).

Fran80. Nissim Francez, "Distributed Termination," *Transactions on Programming Languages and Systems* **2**(1) pp. 42-55 ACM, (January 1980).

Garc81. Hector Garcia-Molina, *Performance of Update Algorithms for Replicated Data*, UMI Research Press, Ann Arbor, Michigan(1981).

Garc84. H. Garcia-Molina, R. J. Lipton, and J. Valdes, "A Massive Memory Machine," *Transactions on Computers* **C-33**(5) pp. 391-399 IEEE, (May 1984).

Good87. Danny Goodman, "The Two Faces of Hypercard," *MacWorld*, pp. 123-129 (October 1987).

Hala87. Frank G. Halasz, Thomas P. Moran, and Randall H. Trigg, "NoteCards in a Nutshell," in *Proceedings of the CHI+GI '87 Conference*, ACM, Toronto, Canada (April 5-9, 1987).

Hala88. Frank G. Halasz, "Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems," *Communications of the ACM* **31**(7)(July 1988).

Horn87. Mark F. Hornick and Stanley B. Zdonik, "A Shared, Segmented Memory System for an Object Oriented Database," *Transactions on Office Information Systems* **5**(1) pp. 70- ACM, (January 1987).

Huan89. Shing-Tsaan Huang, "Detecting Termination of Distributed Computations by External Agents," pp. 79-84 in *Proceedings of the 9th International Conference on Distributed Computing Systems*, IEEE, Newport Beach, CA (June 5-9, 1989).

Jaga90. H. V. Jagadish, "A Compression Technique to Materialize Transitive Closure," *Transactions on Database Systems* **15**(4) pp. 558-598 ACM, (December 1990).

Kapi90. Sarantos Kapidakis, "Average-Case Analysis of Graph-Searching Algorithms," Ph. D. Thesis, Princeton University, Princeton, NJ (October 1990).

Kort88. Henry F. Korth and Gregory D. Speegle, "Formal Model of Correctness Without Serializability," pp. 379-386 in *Proceedings of the SIGMOD International Conference on Management of Data*, ACM, Chicago, IL (June 1-3, 1988).

Lai86. Ten-Hwang Lai, "Termination Detection for Dynamically Distributed Systems with Non-First-in-first-out Communication," *Journal of Parallel and Distributed Computing* **3**(4) pp. 577-599 (December 1986).

Lind81. Bruce Lindsay, "Object Naming and Catalog Management for a Distributed Database Manager," pp. 31-40 in *Proceedings of the 2nd International Conference on Distributed Computing Systems*, IEEE, Paris (April 8-10, 1981).

Lum70. V. Y. Lum, "Multiple-Attribute Retrieval with Combined Indexes," *Communications of the ACM* **13**(11) pp. 660-665 (November 1970).

Mahm76. S. Mahmoud, "Optimal Allocation of Resources in Distributed Information Networks," *ACM Transactions on Database Systems* **1**(1) pp. 66-78 (1976).

Maie86. David Maier, Jacob Stein, Allen Otis, and Alan Purdy, "Development of an Object Oriented DBMS," pp. 472-482 in *Object Oriented Programming Systems, Langauges, and Applications Conference Proceedings*, ACM, Portland, OR (September 9 - October 2, 1986). Also Sigplan notices **21**(11), November 1986.

Mend89. Alberto O. Mendelzon and Peter T. Wood, "Finding Regular Simple Paths in Graph Databases," pp. 185-193 in *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, VLDB, Amsterdam (Aug. 22-25, 1989).

Morr68. D. R. Morrison, "PATRICIA -- Practical Algorithm to Retrieve Information," *Journal of the ACM* **15**(4) pp. 514-534 (October 1968).

Niel90. Jakob Nielsen, "The Art of Navigating through Hypertext," *Communications of the ACM* **33**(3) pp. 296-310 (March 1990).

Orla88. Ratko Orlandic and John L. Pfaltz, "Compact 0-Complete Trees," in *Proceedings of the 14th Conference on Very Large Data Bases*, VLDB, Los Angeles, CA (Aug. 29 to Sep. 1, 1988).

Raym88. Darrell R. Raymond and Frank Wm. Tompa, "Hypertext and the Oxford English Dictionary," *Communications of the ACM* **31**(7)(July 1988).

Robi65. J. A. Robinson, "A Machine-Oriented Logic based on the Resolution Principle," *Journal of the ACM* **12** pp. 23-44 (1965).

Roku88. Kazuaki Rokusawa, Nobuyuki Ichiyoshi, Takashi Chikayama, and Hiroshi Nakashima, "An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems," pp. 18-22 in *Proceedings of the 1988 International Conference on Parallel Processing*, (August 15-19, 1988).

Salt83. Gerard Salton and Michael J. McGill, *Introduction to Modern Information Retrieval*, McGraw Hill Book Company, New York(1983).

Salt88. Gerard Salton, "Automatic Text Indexing Using Complex Identifiers," pp. 135-144 in *Proceedings of the Conference on Document Processing Systems*, ACM, Santa Fe, New Mexico (December 5-9, 1988).

Schw86. P. Schwarz, W. Chang, J. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh, "Extensibility in the Starburst Database System," *Proceedings of the 1986 International Workshop on Object Oriented Database Systems*, pp. 85-92 (September 1986).

Smit86. Karen E. Smith and Stanley B. Zdonik, "Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems," pp. 452-465 in *Object Oriented Programming Systems, Langauges, and Applications Conference Proceedings*, ACM, Orlando, Florida (October 4-8, 1986). Also Sigplan notices **22**(12), December 1987.

Ston83. M. Stonebraker, A. Stettner, N. Lynn, J. Kalash, and N. Guttman, "Document Processing in a Relational Database System," *Transactions on Office Information Systems* **1**(2) pp. 143-158 ACM, (April 1983).

Ston86. M. Stonebraker and L. Rowe, “The Design of POSTGRES,” pp. 340-355 in *Proceedings of the SIGMOD International Conference on Management of Data*, ACM, Washington, DC (May 1986).

Tomp89. Frank Wm. Tompa, “A Data Model for Flexible Hypertext Database Systems,” *Transactions on Information Systems* **7**(1) pp. 85-100 ACM, (January 1989).

Ullm90. Jeffrey D. Ullman and Mihalis Yannakakis, “The Input/Output Complexity of Transitive Closure,” in *Proceedings of the SIGMOD International Conference on the Management of Data*, ed. Hector Garcia-Molina and H. V. Jagadish,ACM, Atlantic City, NJ (May 23-25, 1990).

Wein88. Dale Weinreb, Neal Feinberg, Dan Gerson, and Charles Lamb, “An Object-Oriented Database System to support an Integrated Programming Environment,” *Data Engineering* **11**(2)IEEE, (June 1988).

Wied87. Gio Wiederhold, *File Organization for Database Design,* McGraw-Hill, New York(1987), p. 107.

Woel86. D. Woelk, W. Kim, and W. Luther, “An Object-oriented approach to Multimedia Databases,” pp. 311-325 in *Proceedings of the SIGMOD International Conference on the Management of Data*, ACM (May 1986).