CERIAS Tech Report 2001-81
Indexing in a Hypertext Database
by Christopher Clifton
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Indexing in a Hypertext Database[†]

Chris Clifton[‡] and Hector Garcia-Molina

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

Abstract

Database indexing is a well studied problem. However, the advent of *Hypertext* databases opens new questions in indexing. Searches are often demarcated by pointers between text items. Thus the scope of the search may change dynamically, whereas traditional indexes cover a statically defined region such as a relation. We present techniques for indexing in hypertext databases and compare their performance.

1. Introduction

Hypertext and Hypermedia databases have recently developed along with the technology to store and present more complex information than traditional database records[Chri86, Meyr86]. These systems have also created new ways of accessing data. Traditional databases operate on a query-retrieval basis, where the user provides a query specifying what data is desired. Hypertext systems use a concept of *browsing* and *active objects*, in which the user chooses new data while looking at existing data items.

The first generation of these systems often lack means other than browsing for searching the database. In these systems a "query" is no more than a selection of a reference to another item. In a large database this is not sufficient. Searches may require many repeated user interactions with the system. In addition, the person querying the database may not find the desired information because they do not know *where* to look. The next generation of hypermedia systems must allow queries based on *what* is being looked for. The need for such search techniques has been discussed[Chri85]. However, these techniques should remain tied to the browsing type of searches currently supported by hypertext systems.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

We propose to extend browsing techniques with queries of the form:

In *range* Find *property*.

The *range* consists of two parts: the **start point** of the query, and the **link type** to follow. *Property* is a boolean test on an object, for example checking for the presence of a specific keyword. The corresponding query is similar to what a user would do while manually browsing through hypertext; look at an object (the start point), and follow links searching for objects with the desired property. This allows queries of the form:

> Give me all documents on **sailing** which are referenced by my paper, referenced by papers which I reference, etc.

Here the start point is "my paper", the link type is "references", and the property is that the object is a document on sailing. As another example, a programmer using a software database may want to find all of the uses of a particular variable x in all of the subroutines that make up the programming module currently being examined (here the current module is the *start point*, and we recursively traverse "contains subroutine" links to determine all the modules of interest.) Another query may be to find x in all subroutines of all modules in the system. A third query is to look at all subroutines that are called (directly or indirectly) by a particular subroutine.

What sets these searches apart from traditional database queries is that the *scope* of the query is determined by the data item the user is currently accessing. A hypermedia database can be thought of as a directed multigraph, with the data items as nodes and the links as edges. The scope of a query is the transitive closure formed by following edges of the appropriate type from the start point. Relational database queries, on the other hand, operate on a static scope such as a relation or a set of relations. The scope of a query can be determined from the database

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

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

schema. The queries mentioned above would have to examine all of the subroutines in, say, the subroutines relation; looking for those that make up the particular module or that may be called when the given routine is run. Such information as the calling sequence or nesting structure of the program is difficult to express in a relational query. In contrast, the scope of the hypertext queries is determined by the database content and not the schema. Only the subroutines that are *reachable* (in the desired way) from the point of interest need be examined.

Queries in such a database can be expensive. Some queries on graph-structured data have been shown to be NP-hard[Mend89]. Indexes are used in traditional databases to speed queries by precomputing the results of queries on certain key attributes (such as *keywords*[Baye72, Wagn73, Seli79].) These systems assume that a particular type of search key will be commonly used to access database records. However, in a hypermedia database we also need to know if these data items can be reached by following the appropriate links (such as *references*). This reachability attribute requires special handling, and results in new index structures. In this paper we present methods for indexing in such an environment. These indexes may be thought of as a special type of *multiple attribute* indexing. One of the attributes, such as a keyword, is similar to traditional index keys. The other is specified in terms of a link type, and reflects the "reachability" of objects from the starting point of the query.

Some work has been done on indexing in text databases[Lync88], but this has concentrated on other problems such as keyword extraction and inexact matching. These techniques are useful, and in fact complement the ideas presented here. However, they do not address the problem of indexes whose scope may change dynamically. Work in network databases, such as CODASYL, has also addressed indexing which must respect links in the database[DBTG74]. However, the sources and destinations of links in CODASYL must belong to sets which are determined by the database *schema*. In hypertext systems the use of links is more flexible. The scope of a query cannot always be determined from the database schema; links are part of the data items and may go anywhere and be changed at any time. Another way of looking at this difference is given by the title of Charles Bachman's Turing Award Lecture, *The Programmer as Navigator*[Bach73], in which he discusses network databases and the use of secondary indexes. With hypertext, it is now the *user* who is the navigator. In network databases, the user provides new keys for preprogrammed queries. With hypertext, the user is free to move around the database at will.

A Database Manipulation Language and prototype database manager for such queries has been developed[Clif88]. Objects in the database consist of named attributes (e.g. *key*: cat) and typed links (*reference*: Object D). Some attributes (such as keywords) are short, these are typically used for searches. Other attributes (such as text or pictures) are intended for display only when the desired object (or small set of objects) has been found. Queries in this language include parts which restrict the search based on attribute values (*properties*), and parts which expand the scope of the search by following links (the *range* of the query.) Such a database allows the user to dynamically develop "private libraries" of objects of interest. Queries can then be performed on these libraries exclusive of objects elsewhere in the database (which may match the *property* portion of the query, but not be of interest for other reasons.) The user can return to browsing (and look at information such as pictures) once the "private library" has been restricted to a manageable size. In this paper we will also present results of indexing experiments run on this prototype.

Outline of Paper

The next section discusses the basic algorithms needed to index the type of queries we have outlined above. Section 3 expands this to indexes at multiple points in the database. Section 4 presents an alternative implementation based on a single multi-attribute index. Section 5 gives an analysis of these methods in terms of the time and space tradeoffs of each. This analysis is performed on a regularly structured database we believe to be representative of actual data. We present experimental results on databases with more varied structure in Section 6.

For the bulk of this paper we will assume a tree-structured database. This simplifies many of the algorithms and examples. In Section 7 we will show how these techniques can be expanded to handle databases which are Directed Acyclic Graphs as well as arbitrary Directed Graphs.

2. Single Index

Our indexing technique starts with the simple idea of attaching an index to an object in the database. The index allows lookup of items based on a particular attribute type (the *property* of the query), and covers objects which could be reached from that node following a particular type of link in a "browsing" interface (the *range* of the query.)

What is indexed

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

Figure 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. However, since it is not **reachable** from *root* via solid links, it is not in the index.
- Item *G* is in the index, even though its parent (*C*) does not appear in the index. Node *C* is in the *scope* of the index, but does not appear since it has no *key* attribute.

The index of Figure 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 system.

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. However, certain special information is required. In addition to pointers from the index to relevant objects, objects will be required to have back pointers to indexes which *potentially* include them. This is necessary in order to properly maintain the index. For example, in Figure 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 index updates.

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 a hypertext system, determining what indexes a data item is in may be as difficult as building the index (in terms of number of items referenced.) The use of back-pointers is necessary to maintain the indexes at a reasonable cost when data items are modified. In addition, when a data item is added to the database the indexes which 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 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 algorithms 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.

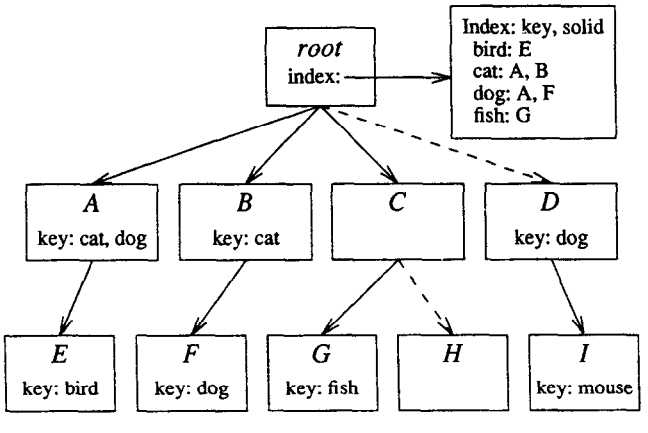

Figure 1: Index of a tree-structured database.

```
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 ∪ 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.*
$\forall$ *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 which is not indexed can still make use of indexes. The simple case is making use of an index which is associated with a node which is reached at some point in the search. This is already done in the above algorithms. However, 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 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 which 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 Indexes

In a real system, there may be many nodes from which we often make queries. We could build an index at each of these nodes, but this leads to space problems due to replication of information. Figure 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 which is below *A* must have back-pointers to both indexes.

Eliminating Replication

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 would still end up requiring more space than the data itself. In addition updates to the database may take a long time because they must modify many indexes.

This replication can be eliminated by requiring indexes to refer to "lower" indexes, rather than directly indexing the entire subtree. This is illustrated by the indexes pointed to by **dotted** lines in Figure 2 (just the ones on the left side of the Figure.) A search for all items in the database (starting at *root*) which 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

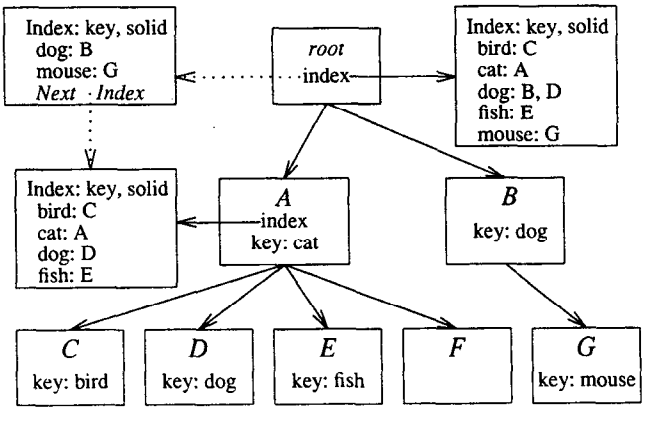

Figure 2: Tree-structured database with two indexes.

benefits of indexing. However, we expect the typical cost will be much smaller. This will be discussed in Section 5.

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. However, changing the pointers 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 which 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 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.

4. Single Multiple-Attribute Index

An alternative to the previous structure is to use a single database-wide index for each type of key. In a sense this is a multiple attribute index[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 which have indexes) to the objects that can be found from those anchor points. The structure of the primary and secondary indexes could be any of a number of things, including B-trees, hash tables, sorted lists, etc. A naive implementation of the secondary indexes, in which each anchor point hashes to a list of all of the objects reachable from that anchor point, could require $O(n^2)$ space per secondary index (where n is the size of the database). However, all of the objects at many anchor points are reachable from other anchors (e.g. in Figure 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 3 is a sample index containing entries for a few keywords based on the database of Figure 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* which contain the keyword "dog". We use the primary index to find the secondary index associated with "dog". We also need all of the anchor points reachable from *root* (done using the reachability graph, these are *root* and *A*.) Next we find all of the objects reachable from these anchor points using the secondary index. The objects *B* and *D* are the result of our search. More formally, the **Find** algorithm is:

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

  Note that the previous steps can occur in parallel.

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

As written this assumes that the current node has an index.

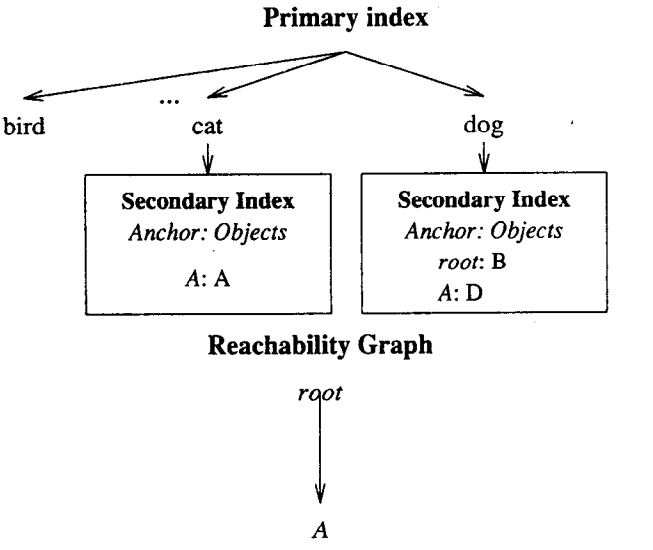

Figure 3: Single Multiple-Attribute Index.

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 would take quadratic space (and is also expensive to compute[Ullm90].) However, for a tree-structured graph (or tree-structured parts of the graph) reachability can be expressed 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 had to construct a new index for each type of link. However, in this case we may reuse the primary index. 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 which 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 and 3) 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.

5. Cost Comparison

The methods of indexing we have introduced (single indexes, indexes with replication, indexes without replication, and multiple-attribute indexes) each have advantages and disadvantages. 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 results in an analysis too complex to be included in this paper. 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 4.
- 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

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

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:

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

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:

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

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 4 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.* Finding the appropri-

* 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.

ate 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:

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

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:

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

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 $(B^j+1)(1-(1-P)^{B^{j-1}})$, 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:

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

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 page 6. 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 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 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 4. The gains provided by indexing are substantial.

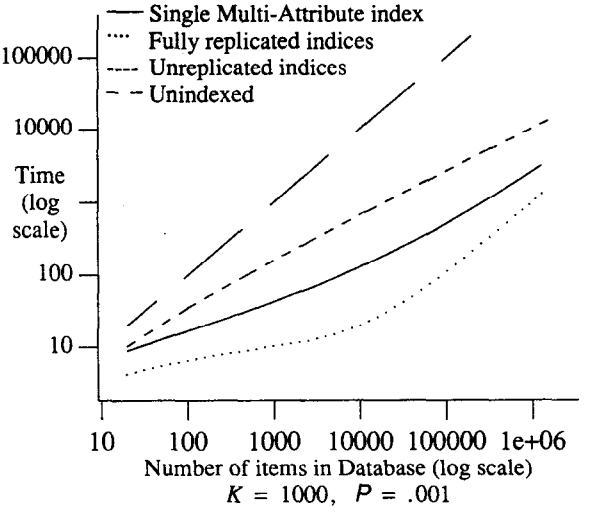

Figure 4: Find Time vs. Database size, search from root.

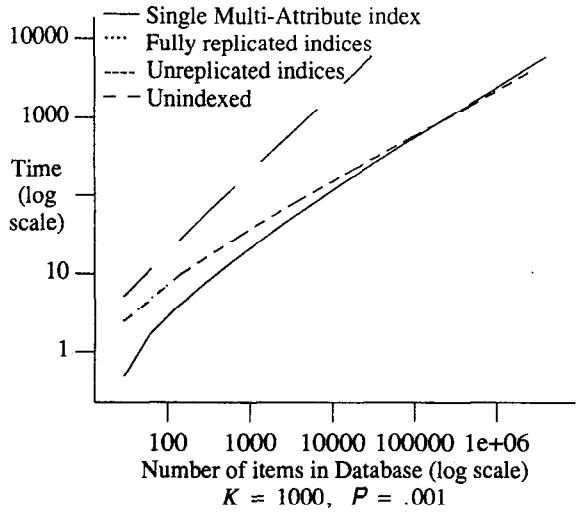

Figure 5: Find Time vs. Database Size, just below root.

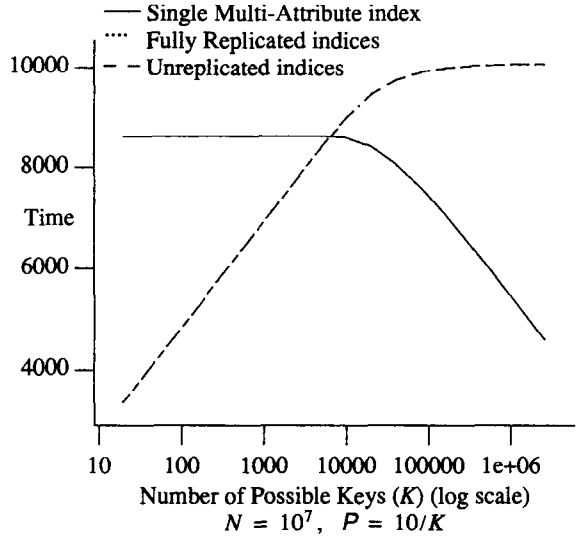

Figure 6: Find Time vs. Number of Keys, just below root.

Figure 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 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 7 is space versus number of items in the database. We have assumed that $c_s = c_r = c_p = 1$ 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 8 corresponds to Figure 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

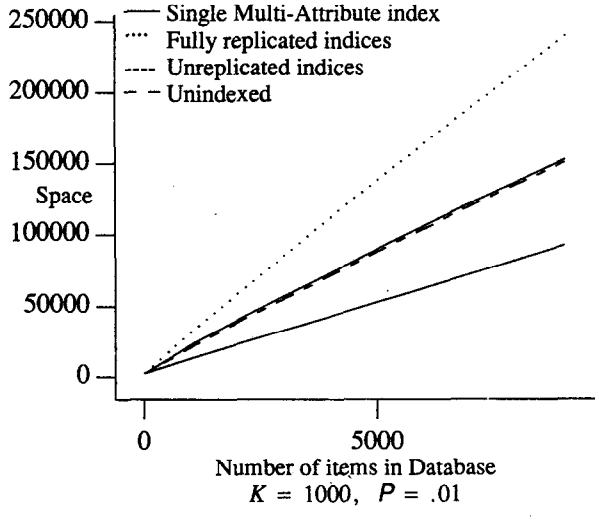

Figure 7: Index Space vs. Database Size.

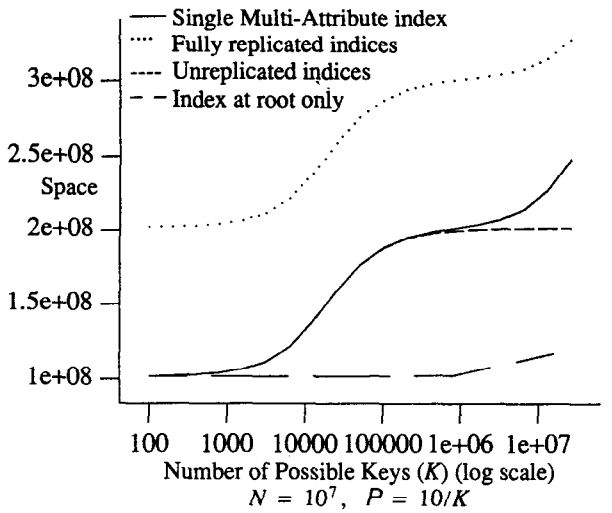

Figure 8: Index Space vs. Number of Keys.

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.

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.

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 which are dominated by reads, so we have concentrated on the search times. However we do feel it is important to say something about the costs of building and updating indexes.

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 which 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. However, if a major portion of the graph is changed, 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 which 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.

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.

6. 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 some experiments using a prototype query processor/main memory database on less regularly structured databases to verify this. We include graphs in this section which plot the experimental results alongside predicted results from the analysis of the previous section.

The prototype query processor is written in an object-oriented language (Eiffel) and runs on a DEC 5410.

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 which 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 page 6 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 $E(n) = n \cdot 3ms$. 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 $T(n) = \log_2(n) \cdot 1.5ms$.

In order to see how well our analysis predicts performance on databases without a regular structure, we performed experiments on randomly constructed databases. Note that the databases used in the experiments are not entirely random collections of nodes and links. We expect large hypertext databases to have a structure which resembles a tree more than, for example, a completely connected graph. Therefore our experiments are based on data with a somewhat regular structure. We constructed two types of databases, trees and Directed 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

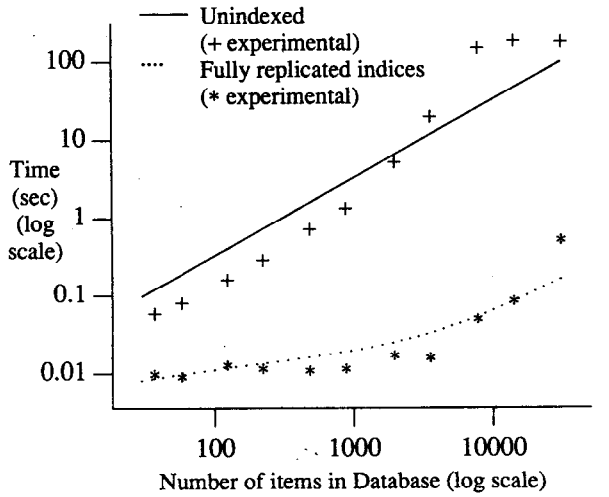

Figure 9: Queries from root, tree database.

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 9 gives results of F_1 queries (searches from *root*) performed on a tree-structured database built to the above constraints. Figure 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 which were not tree-structured. To the databases used for Figures 9 and 10 we added links which 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 11 and 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 do appear to be in the ballpark, particularly with the larger databases.

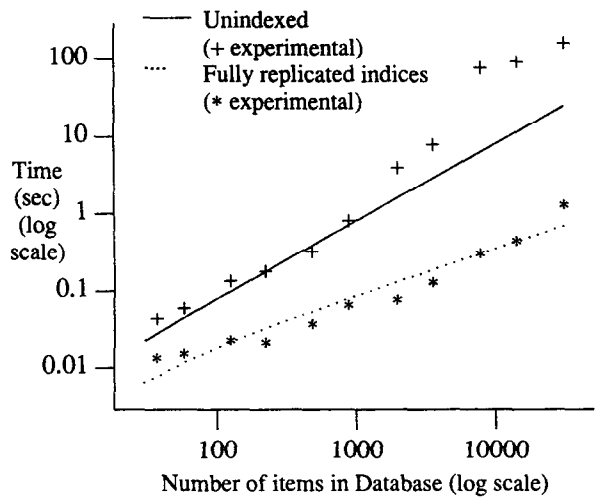

Figure 10: Queries from just below root, tree database.

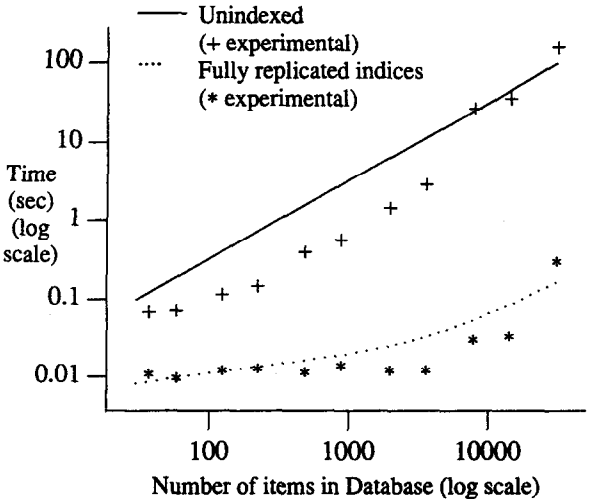

Figure 11: Queries from root, DAG database.

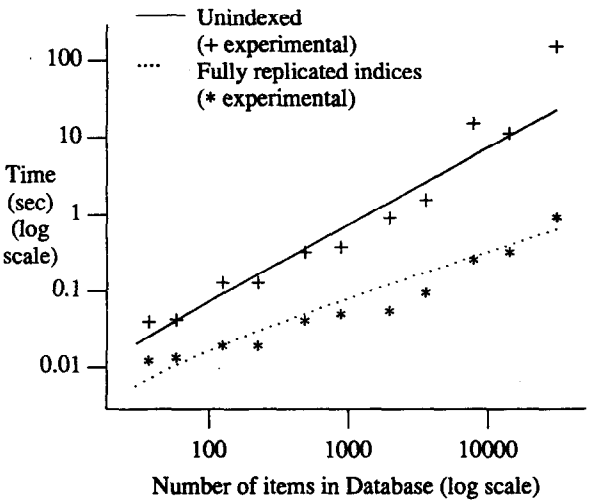

Figure 12: Queries from just below root, DAG database.

The trends in the experiments coincide relatively well with the predictions from the analysis. The model we developed in Section 5 cannot be used to predict the exact performance of indexes on a particular database. However, the model can be used to study tradeoffs and general trends.

7. Graph Structured Databases

The previous algorithms have been presented in the context of a tree-structured database. Directed Acyclic Graphs and arbitrary Directed Graphs present new problems. Figure 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

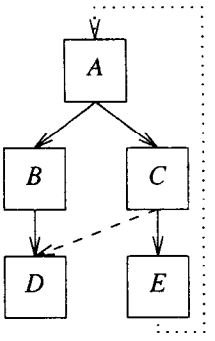

Figure 13: Arbitrary directed graph database.

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. However, link deletion becomes more difficult. This is not a serious problem, as it is related to Garbage Collection, which has been addressed extensively[Cohe81]. We will briefly summarize some possible solutions.

Directed Acyclic Graphs

In this case, 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. By directly, we mean that the reference count of an object is the number of parents it has which 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 is 0 is the delete performed.

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.

8. Conclusions and Further Work

Technology is providing us with the capability to greatly increase the amount and kind of data we store electronically. However, the speed of accessing this storage is not keeping pace with its size. For example, the IBM 3380 Model AD4, a high performance magnetic disk, can store 2.5 Gigabytes and provide this to the user at up to 3 MB/second[IBM86]. The Kodak 6800 Optical Disk System stores 6.8 Gigabytes[Kod87a]. Combining this with the 6800 automatic disk library, an optical disk jukebox, provides 6.8 Terrabytes in similar floorspace to the IBM 3380. However, the access rate is considerably slower; the basic access rate is 1 MB/second, and "seek" times (if a new disk must be loaded from the jukebox) are on the order of 2 seconds[Kod87b]. As a result indexing is becoming increasingly necessary; searching through even a small percentage of the database is unreasonable.

With such storage capacity available, people will want to save data which does not mate well with traditional database systems. *Hypermedia* systems provide a nice paradigm for much of this information. We have presented some techniques for indexing which work with the type of searching done in hypermedia browsing systems.

Alternative Methods

Alternatives to the indexing methods presented here have been explored. They have all turned out to have significant drawbacks. Some alternatives that were considered were:

- An index on a key would return an ordered list of items containing that key. Stored with the *anchor point* for the index would be an ordered list of objects reachable from that index. These lists would be merged to obtain the intersection, thus giving the reachable objects containing the desired key. We developed a compact representation for this list of reachable objects, giving a space slightly better than the single multiple-attribute scheme we have presented. However, in many cases a query may cover a substantial portion of the database. Thus the list of reachable objects would be long, and the merge would take time on the order of the length of this list. The result is that the index find is considerably slower than the single multiple-attribute case.
- One fix for the above method is to associate a hash table with each anchor point, rather than an ordered list. This would allow the find to proceed in time proportional to the total number of items in the database which matched the key, which is likely to be a relatively small number. The problem with this is space: The size of each hash table would be proportional to the size of the graph reachable from that anchor point. This is comparable to using fully replicated indices; however fully replicated indices are considerably faster.

Other Applications

There may be uses for this style of indexing in applications other than hypertext. Similar queries may arise in object-oriented databases. These systems have been proposed as a likely base for hypertext databases[Woel86]. The indexing methods presented here can be applied in any system where pointers between data items are important to (and possibly implicit in) queries.

Federated databases[Heim85] may also provide an application for these indexes. Although indexing at the local level is the responsibility of the individual databases, global indexing could make use of some the techniques of

"hierarchies of indexes" discussed in Section 3. Distributed databases may pose similar problems.

This type of indexing may apply to nested relations[Dada86, Fisc85] as well. This depends on how flexible the nesting is; with enough constraints traditional indexing techniques could be used.

There is still considerable work to be done in this area. We have discussed *how* to index, but left open the question of *what* should be included in the index. With increasingly unstructured data, such as text, being included in databases, some automatic indexing methods are needed. Work has been done in this area[Salt88], automating such tasks as "back of the book" indexes. These techniques should be incorporated in the construction of large-scale databases.

Acknowledgements

Some of the ideas behind the database and query language which motivated this work were initially developed at Xerox P.A.R.C. in discussions with Robert Hagmann, Jack Kent, and Derek Oppen. We would like to acknowledge their contribution.

References

[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).

[Bach73] Charles W. Bachman, "The Programmer as Navigator," *Communications* **16**(11) pp. 653-658 ACM, (November 1973).

[Baye72] R. Bayer and C. McCreight, "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica* **1**(3)(1972).

[Chri85] S. Christodoulakis, "Multimedia Data Base Management: Applications and Problems. A Position Paper," pp. 304-305 in *Proceedings of SIGMOD '85*, ACM (May 1985).

[Chri86] S. Christodoulakis, M. Theodoridou, F. Ho, M. Papa, and A. Pathria, "Multimedia Document Presentation, Information Extraction, and Document Formation in MINOS: A Model and System," *Transactions on Office Information Systems* **4**(4) pp. 345-383 ACM, (October 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).

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

[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).

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

[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).

[Fisc85] Patrick C. Fischer and Dirk Van Gucht, "Determining When a Structure is a Nested Relation," in *Proceedings of the Eleventh International Conference on Very Large Data Bases*, VLDB, Stockholm, Sweden (August 21-23, 1985).

[Heim85] Dennis Heimbigner and Dennis McLeod, "A Federated Architecture for Information Management," *Transactions on Office Information Systems* **3**(3) pp. 253-278 ACM, (July 1985).

[IBM86] *IBM 3380 Direct Access Storage General Information*, International Business Machines Corporation(March 1986). Publication #GC 26-4193-2.

[Kod87a] *Performance Specifications for the Kodak 14 inch Optical Disk Recorder*, Eastman Kodak Company(January 1987). Publication No. PS-XXXX, preliminary.

[Kod87b] *Performance Specifications for the Kodak Optical Disk System 6800 Automated Disk Library Models A and B*, Eastman Kodak Company(April 1987). Publication No. PS-0714-1, preliminary.

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

[Lync88] Clifford A. Lynch and Michael Stonebraker, "Extended User-Defined Indexing with Application to Textual Databases," in *Proceedings of the 14th Conference on Very Large Data Bases*, VLDB, Los Angeles, CA (Aug. 29 to Sep. 1, 1988).

[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).

[Meyr86] Norman Meyrowitz, "Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework," pp. 186-201 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.

[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).

[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).

[Seli79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond L. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System," pp. 23-24 in *Proceedings of the SIGMOD International Conference on Management of Data*, ACM (1979).

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

[Wagn73] R. G. Wagner, "Indexing Design Considerations," *IBM Systems Journal* **12**(4)(1973).

[Woel86] D. Woelk, W. Kim, and W. Luther, "An Object-oriented approach to Multimedia Databases," pp. 311-325 in *Proceedings of SIGMOD '86*, ACM (May 1986).