THE DESIGN OF A DOCUMENT DATABASE

Chris Clifton
Hector Garcia-Molina
Robert Hagmann

CS-TR-177-88

September 1988

(Appeared in the ACM Conference on Document Processing Systems,
Santa Fe, New Mexico, Dec. 5-9 1988.)

The Design of a Document Database[†]

Chris Clifton
Hector Garcia-Molina
Princeton University
Robert Hagmann
Xerox Palo Alto Research Center

Abstract

In this paper a Document Base Management System is proposed that incorporates conventional database and hypertext ideas into a document database. The Document Base operates as a *server*; users access the database through different *application* programs. The query language which applications use to retrieve documents is described.

September 13, 1988

The Design of a Document Database[†]

Chris Clifton
Hector Garcia-Molina
Princeton University
Robert Hagmann
Xerox Palo Alto Research Center

1. Introduction

Conventional database systems evolved as a replacement for data stored as individual records in a file system. Systems such as INGRES[1] and System R[2] provide for data integrity, sharing, security, and efficient searching and information retrieval. However, these systems are all based on fixed-length, highly structured records of information. Not all information can be conveniently represented with such a model[3]. In particular, *documents* must be represented as unstructured, uninterpreted strings, making it difficult to capture their structure, properties, and interrelationships.

Our objective is to implement a *Document Base System* that provides for integrity, sharing, security, and efficient searching of documents. The *document model* we propose for this system is general enough that is supports a hypertext view of documents (links and nodes), as well as traditional linear documents. It is flexible enough so that documents can be source programs, circuit layouts, diagrams or pictures. The system has the searching and retrieval capabilities of traditional information retrieval systems (e.g., keyword searches), as well as database facilities such as transactions, versions, and data dictionaries.

We see documents as being composed of both structured and unstructured information. The structured information represents the properties of the document and can be used for searching. Examples of structured information are keywords, names of authors, titles, references to other documents (links), source language (e.g., LISP, C), technology (e.g., NMOS, CMOS), and so on. Unstructured information, such as basic text and diagrams is stored and retrieved by the system but is not used in searching. Incidentally, the representation of text as partially structured information has been explored in the Information Lens project at MIT[4], and has been found to be a useful tool from the human viewpoint. We believe that considering documents to be partially structured is useful for database design as well.

In many cases we expect the document base system to run on a separate machine or environment from the end user. Thus, our view is that multiple applications at various workstations send requests to a *document server*. In light of this, one of the key requirements for our system is that it have a powerful non-navigational *Document Manipulation Language* (**DML**) that reduces the number of user-system interactions. For instance, suppose that a user has identified a set of documents of interest. The user would now like to find all documents that are referenced in the bibliographies of the initial set of documents that have the keyword "cat" occurring within 50 positions of the word "dog". However, only the abstracts of these documents are to be retrieved. We would like this type of request to involve a single interaction with the system. The user sends the DML query, the system searches (possibly) many documents, and returns only the relevant data. For the search, the system can use auxiliary search structures (e.g., keyword indexes). This is

[†] 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.

analogous to the way conventional relational systems operate: the user describes the data of interest, and the system determines how and what to search. The strategy is very different from that used in most hypertext systems, where each link has to be explicitly explored under control of the end user. Our approach permits more efficient searches, reduces the amount of data transferred to the user, and cuts down the number of requests.

It is important to note that the DML is not a full programming language. In this we are following the precedents of database languages such as SQL[5] and QUEL[6]. Applications that access documents will be written in a conventional (host) programming language with embedded DML statements. This provides for a clean break between document retrieval tasks which should be done at the server and presentation issues which are handled by the application. It also provides a degree of security at the server, since running arbitrary user code at the server would be dangerous. We stress that the DML is not a human interface. Humans interact with the application running at their workstation, which in turn makes the necessary DML calls.

In this paper we present the document model and manipulation language. Because of space limitations we will simply highlight the main features and give representative examples. After describing the DML, we will very briefly discuss other related aspects of our system such as indexes, version management, memory organization, and triggers. Finally, we also include a short section listing some of the previous work in this area.

2. Document Model

A document 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", "Text goes here...")
 (contents, <pointer to a document>, 1)
 (contents, <pointer to a document>, 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 of type string, with key probably being a short string of fixed length. This definition of the triple type "string" is stored by the system is a type table. Some of these types are defined by the system, but other definitions can be added by users. Type definitions extend across the system, which encourages the sharing of data between applications. (Structured type names, such as Hector/keyword and Chris/keyword, can be used to allow different users to use the same name for different purposes if desired.) The primitive types that are allowed for key and data items are discussed later.

The triple type "keyword" specifies that the triple contains a document keyword (short string) and its position in the document (integer). This triple type is recognized by the system for building keyword indexes. Note that the system treats the position field simply as an integer, and it is up to the application to interpret this position as bytes or words within the document. These keywords probably appear somewhere in the "text" triple, but the system is not aware of this. It is up

to the application to maintain the consistency of the text and the keywords. (Note that there may be types in addition to keyword that have associated index structures.)

The document 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 the semantics of each document property cannot be understood by the system. A second reason is efficiency. If the document server is to efficiently examine large numbers of documents, the properties used for searching must be simple and compact. A third reason is that the complexity of the DML 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 document model. However, in spite of the model simplicity, we believe that it is sufficiently powerful.

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 and unstructured data field with all the information encoded in it. 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 a document in itself. In this case, the original document contains a simple pointer to the link document. The link then contains the relevant properties (e.g., date, name, color, etc.) as well as one or more pointers to other documents. In summary, applications that require a richer structure than what is provided by the basic model can provide it for themselves. (Incidentally, our system does provide other primitive pointer types: version pointers and back pointers. These will be discussed later.)

Note that documents are represented as sets, so triples are not ordered within a document. This restriction substantially simplifies our language. Ordering can be obtained by linking the components together (e.g., part A points to part B which points to part C). As an alternative, ordering can be indicated by the data field, as illustrated by the last two triples of our sample document.

3. Document Manipulation Language

The DML is used to represent queries. The queries we wish to support fall into two primary types:

- Document retrieval along pointer chains. This is important both for references and for retrieving parts of documents. These queries are the major difference between *hypertext* and conventional databases.
- Searches for documents 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 documents (particular patterns of pointers to other documents.)

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

Most query operations take a document (or set of documents), and return a new document (or set) without modifying the original document. Changes to a document are made with functions which operate on a single document.

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

3.1. Set operations

Since documents are structured as sets, the basic set operations, *union* ($\cup$) *intersection* ($\cap$) and *difference* ($-$) are provided. Each binary operator takes two documents, and returns a new document (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. As discussed above, A and B are variables in the host language, not the actual documents. The statement

```
A ∪ B → B
```

leaves B pointing to a new document which is the union of the documents (sets of triples) originally pointed to by A and B. The original documents are not destroyed. This newly created document is temporary (it will disappear when the application terminates) unless another document points to it. Creating pointers and adding them to documents will be discussed later.

3.2. Basic filters

These are operations which take a document, and return a new document which may include a subset of triples or modified triples of the original document. They operate by looking for particular triples, primarily based on the triple-type and key, and performing an operation such as adding the triple to the document being created.

Filters are based on triple selection using pattern matching. Perhaps it is easiest to start with an example. Given a document (document id) D, we can construct a new document consisting of just the authors of the original document as follows:

```
D(string, "author", ?) → 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*") )
  → document id
```

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

3.3. Sets of Documents

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

Another advantage of queries on sets is that it gives the user an idea of the complexity of a query. The time required to execute a query is directly related to the number of documents in the queried set. Thus the user has some control over and knowledge of the time a query will take based on the size of the set queried.

Sets are actually a type of document. This is done using triples containing pointers. A set is simply a document containing pointers to other documents. Document S in Figure 1 is an example of a set containing the documents 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 a document.
- Sets can be permanent, in the same manner as a document is made permanent. This allows users to build "private libraries".
- It is easy to build annotated bibliographies; since a set is a document, associating text, keywords, and other information with it is simple.

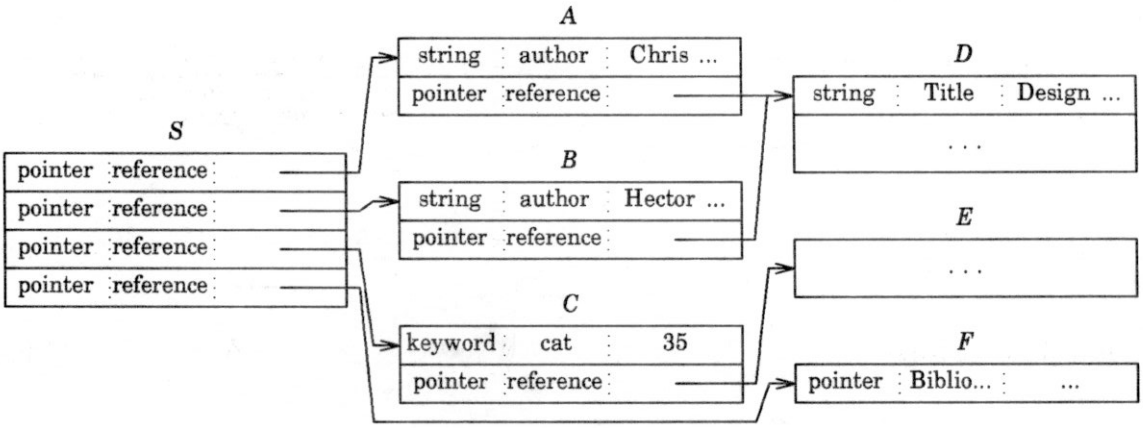

Figure 1. Set S containing documents A, B, and C.

- A paper which contains references can also be used as a set *of the referenced documents.* This allows easy "literature search" operations.

The set operations mentioned above 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 a document in the set, $S \cup T$ produces a new set of triples which points to all of the documents in either S or T. In fact the primary use of these operations is likely to be on documents which are considered to be "sets of documents" rather than on individual documents.

3.4. Set filters

Set filters are queries used to select documents meeting particular criteria. For example, we may wish to find all documents by a particular author, or all documents which reference a given paper. These operate by selecting documents out of a *set*, rather than the entire database.

Basic filters select individual triples from a document based on the properties of those triples. With sets, we want to select documents from the set based on properties of the document pointed to. For example, in Figure 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 which are similar to the single document filters discussed above. As an example, to select those documents from the set (document of pointers) S which were written by either Chris or Hector we could use

```
S | ( (string, "author", "Chris*") OR
      (string, "author", "Hector*") )
  → document id
```

An equivalent statement would be

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

Wild cards (?) can also be used here.

Some sample queries are:

Select documents in S with keyword "cat" and place them in document T.

```
S | (keyword, "cat", ?) → T
```

Select documents with keyword having

prefix "ca" .

```
S | (keyword, "ca*", ?) → T
```

Select documents having keyword matching "?a?" occurring after the 30th word in the document.

```
S | (keyword, "?a?", >30) → T
```

Select documents with either "cat" or "dog" as a keyword.

```
S | (keyword, "cat", ?) OR
    (keyword, "dog", ?) → T
```

Select documents having "Princeton" as a keyword or in the title.

```
S | (keyword, "Princeton", ?) OR
    (string, "title", "*Princeton*") → T
```

Select documents having both "cat" and "dog" keywords.

```
S | (keyword, "cat", ?) |
    (keyword, "dog", ?) → 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".)

3.5. Matching variables

Related to wild cards are *pattern matching variables*. These are wild card characters which must match at various points in an expression. As an example, let us select documents from the set S with "cat" and "dog" keywords within 10 words of each other:

```
S | (keyword, "cat", ?X) |
    (keyword, "dog", |X−?|<10) → T
```

In the first filter, all values of the position of "cat" in each document are found. The second filter is satisfied if there is a "dog" within 10 words of any "cat" *in the document under consideration.*

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

Some more complex examples using set filters and matching variables are:

Select documents with at least two occurrences of "cat", at least one of which is after position 27.

```
S | (keyword, "cat", ?X) |
    (keyword, "cat", X≠?Y ∧ Y>27) → T
```

Select documents with second "cat" after position 27.

```
S | (keyword, "cat", ?X≤27) |
    (keyword, "cat", ?Y>27) |
    not (keyword, "cat", <X) |
    not (keyword, "cat", X<?<Y) → T
```

The actual semantics of pattern matching variables are similar to Prolog unification[7]. Their use is also related to *joins* in a relational database. 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. Another way of thinking of them is that each instance of a document passing through a filter has its own set of pattern matching variables. Each variable is actually a set of values which it matches in that document. An expression using

the variable is true if any of the values in the set would make the expression true.

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. However, the data must be available to the host language. This is done using pattern matching variables, which can be defined as accessible to the programming language. The operators used are →X and ←X, where X is defined in the programming language. The → causes the value from the triple to be placed in the variable. This binding holds for the execution of set of code bound to the DML expression. If multiple values exist which match the variable, the code is executed once for each possible binding. This is similar to the manner in which conventional embedded database languages operate. As an example, to print the authors of document *D* using a routine *display_author* written in the host language, we write:

```
D(string, "author", →X)
    { display_author(X) }
```

The ← causes the matching variable to be bound to the current value of the same variable in the program. ←*X* occurs in place of ?*X* in a query.

3.6. Pointer operations

In order to allow following of pointers, two *dereferencing* operations are provided. These are ↑*X* and ↑↑*X*, where *X* is a matching variable. The first is a simple dereference; it returns the document pointed to. The second gives both the document pointed to and the original document. These are best shown by example:

```
S | (pointer, "reference", ?X) | ↑X → T1
```

produces the documents referenced by documents in *S*. T_1 is itself a set (document of pointers), and can be operated on in the same manner as *S*. Looking back at Figure 1, we see that T_1 contains *D* and *E*. If we wish to retain the pointing documents we use a ↑↑:

```
S | (pointer, "reference", ?X) | ↑↑X → T2
```

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 Figure 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.

Some more examples are:

Place documents bibliographically referenced by documents in S into T.

```
S | (pointer, "Bibliographic", ?X) | ↑X → T
```

Documents referenced with either Bibliographic or foo references.

```
S | (pointer, "Bibliographic", ?X) OR
    (reference, "foo", ?X) | ↑X → T
```

References of documents with keyword "cat".

```
S | (keyword, "cat", ?) |
    (pointer, "reference", ?X) | ↑X → T
```

Documents which are references of documents in S, where the referenced document has the keyword "cat".

```
S | (pointer, "reference", ?X) | ↑X |
    (keyword, "cat", ?) → T
```

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

```
D (pointer, "reference", ?X) | ↑X → T3
```

produces the documents referenced in D. Note that T_3 is a set. Since a set of documents is simply a document containing pointers, the above statement simply strips non-pointers from D. An equivalent statement would be

D (pointer, "reference", ?) → T_3

3.7. 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 implementation will be clever in processing such a query and in fact only process each document once.

If we want to find all documents within a tree rooted at the current set, we can use the * operation:

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

Note that this repeats the operation in brackets until the result reaches a fixed point;

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

would return the empty set, as it would continue until there were no more referenced documents. Also note that as defined, this last query would not terminate if there were a cycle of references. Such situations will be detected and an error returned.

A simple implementation of iteration, merely repeating the given query until no new documents are found, would result in documents being needlessly being reprocessed. The reprocessing is inefficient but does not produce incorrect results, even if dereferencing is part of the query. To eliminate reprocessing altogether, we can associate an "already processed by this iterator" mark with each document while the query is being processed.

In addition, a level number (based on the number of iterations of the query) can be attached to each document. De-referencing operations will copy this level number (but not the already processed mark) to the new document. As each document finishes the query, this information can be used to determine if it should be run through the query again (no already processed mark and level number less than the number of iterations desired, in which case the mark is set and level incremented) or passed on to the next part of the query. Note that this information can be placed with other such run-time data, such as possible values for matching variables.

This implementation would allow iteration to be done with only a single pass over each document which is potentially in the result. In fact most of these queries require only a single pass over the documents in the original set, or in the case of dereferencing, over the descendants of that set. In

addition, this design allows for easy extensions to a parallel implementation.

3.8. 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_document() → *document 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.

A copy operation is also provided. The use of **copy** is:

copy_document(*document id*)
→ *new document id*

The *delete* operation removes a document from the database.

delete(*document id*)

Attempts to reference a deleted document will be detected and an error will be returned. After further study we may choose garbage collection over explicit deletion.

There are also operations which 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 also be added; currently this will be done using delete and add.

4. System description

As mentioned earlier, this paper primarily discusses the Document Manipulation Language. Designing a document base poses a number of other problems. Some of our ideas on these issues are listed here.

4.1. Basic data types

We propose a limited type system. This is less general than, say, an object oriented system. The limited domain of applications (documents), however, reduces the need for generality, and there are advantages in efficiency and ease of prototyping which result from having such a type system. Unstructured data types are provided for applications which require them. The document base manager will not be able to perform as many operations directly on these types, however.

The operations on these types are not defined in this paper. In most cases, the appropriate operations are obvious. The interface to the host programming language is similar to that for the basic DML operations described previously.

4.1.1. Short (fixed-length) types

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

- *Words* are short character strings. Typically uses are for keywords or index entries. An idea for an efficient fixed-length representation of **words** is mentioned later.
- *Numeric* data types (including dates, times, reals, etc.) and appropriate numeric operations will be supported.

- *Pointers* are perhaps the primary thing which sets hypertext apart from conventional documents. 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)
```

The pointers we have described up to this point are simple pointers to documents, uninterpreted by the system. Our system will provide at least two stronger types of pointers. The insertion of a triple of the form (strong-pointer, key, <pointer>) in a document D will force the insertion of a triple (back-pointer, key, <pointer to D>) in the referenced document. The system will not allow the deletion of the referenced document unless the strong-pointer at D is first deleted. Note that the pointer and back pointer within these triples are plain document pointers. The additional semantics of strong pointers are enforced at the triple level.

To support document versions, the system provides some basic support. (More elaborate mechanisms can be built on top of this.) When the triple (version, <null pointer>, timestamp) is inserted into a document (with timestamp set to the largest possible value), this indicates that the system will keep a linked list (with back pointers) of the versions of this document. Each update to the document creates a new version. The timestamp of the older version is set to the time of the update, and a pointer to the newest version is stored automatically. Thus, the tail of the list is the latest version.

A pointer of the form (latest-pointer, key, <pointer to D>) is followed by the system by going to document D, and then following the version chain to most recent version of the document. Hence, even though D points to an older version, the query that follows this pointer always gets the latest version. A conventional pointer (pointer, key, <pointer to D>) will still get to the older version.

4.1.2. Long (variable-length) types

These are intended for use primarily in the **data** field. Although of variable length, most will be relatively compact, the exception being *text*. In our prototype, all of the document except for text data items will be cached in memory. This speeds queries which do not require looking at text items.

- *String*s are short sequences of characters; at most a sentence. Standard string operations will be allowed, but they will be slower than equivalent operations on **word**s.
- *Block* is an unstructured type, where the data is of a small (although not necessarily fixed) size. Searches will be allowed on this data by allowing the application to provide (restricted) functions to determine matches.
- *Text* is the basic unstructured type. Any operations which must be performed on text blocks (other than creation and deletion) must be provided by the application. This is not only a medium for the written word; text blocks can be used for pictures, executable code, or other data which does not fit into the normal type system. In our prototype, text will be stored as a file.

4.2. Triple types

These are the actual key-data combination types, which can be defined by applications. In order to ease the problem of type definition, each database will have a "reserved" catalog document. This will contain 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 will be provided. Although this does not automatically resolve conflicts in the use of triple types, it does simplify human resolution of the problems.

4.3. Indexes

Our system will initially provide inverted-list keyword indexes (other indexes may be added later.) Each index will be associated with a *base document* D and a pointer key label L. The index at D will serve all documents that are reachable from D via pointers with key L. The L pointers are restricted to form a tree. Each document in the tree will have a back pointer (with key "back-index") to the index.

As an example, authors may wish to look for a particular keyword only in documents they have written. A global database index would probably not be useful here. However, if each of the author's documents are pointed to by a "root" set for that author, an index at that root would be appropriate. Of course, each document would have to contain pointers with the appropriate key L to all sub-parts of that document. This would probably be done as part of a "document writing" application, and the indices and pointers would be invisible to the end user.

Keyword queries that involve D or its descendants will utilize the index. (Taking full advantage of an index is a challenging query optimization problem that we have not addressed yet.) When a document with back index pointers is modified, the appropriate index (or indexes) must be updated if necessary. In particular, if an L pointer is cut, all the documents down the tree must be removed from the index.

It is also possible to have indexes at several levels within the same tree, with each index serving its own sub-tree. Another challenging problem involves taking advantage of sub-indexes within a tree to reduce the size of a higher level index.

4.4. Representation of Words

The **word** data type is intended as a search key. In order to perform these searches more efficiently, we will use a fixed-length representation for this type. A structure based on the patricia tree[8] can be used to map *keyword*→**Z**. Assigning numbers alphabetically allows for simple lexical comparison. Using 32 bits will give a sparse mapping, allowing the addition of new words without changing the old mapping or interfering with the alphabetical property of the map.

4.5. Transaction management

As in any database, transactions are of some importance. However, documents may require new types of transactions; serializability may not be appropriate. Some form of locking must be done, this could be based on an explicit check-out mechanism.

Under consideration are alerters and triggers, actions which are started when another action happens in the database such as reading or writing a document. The difference is that alerters are "soft", in the sense that they do not have to happen in as timely a fashion as triggers. The exact definitions of these are related to transaction management, and further specification must wait on a better understanding of transactions.

4.6. Massive Memory Machine

Our Document Database Management System is being undertaken as part of the Massive Memory Machine Project (MMM) at Princeton[9]. The document system will be implemented on a prototype with one gigabyte of main memory. Our strategy will be to cache in memory only the structured portion of documents, leaving text data on secondary storage. Since our document model is so simple and regular, we expect to be able to store the in memory portion of the documents very compactly. This in turn means that for many queries all of the search data (e.g., keywords, pointers) will be memory resident, making it possible to examine vast numbers of documents. Once the desired documents are identified, accesses to secondary storage would be needed to retrieve the bulky parts such as contents of a paper, pictures, etc.

Simply to illustrate, suppose that in-memory part of documents is on the average 500 bytes. (At 10 bytes per triple, this allows for say 20 pointer triples, 20 keyword triples, and 10 other triples). A one gigabyte memory can then hold about 2 million documents. As memory prices continue to drop and memories grow, this number will also increase. We would expect a cache that holds the few million most frequently accessed documents to reduce the number of secondary store accesses very significantly, especially for queries that must traverse and examine large numbers of documents (e.g., searching through a large library).

5. Other Systems

A number of hypertext systems, such as Notecards[10], Hypercard[11], Textnet[12], Intermedia[13], and MINOS[14] have been developed. These systems provide for a new means of representing information. The presentation of information, in some cases incorporating a variety of media, is impressive. However, the structure of information in some of these systems bears little resemblance to traditional documents. The emphasis has been on the user view of a single document, as opposed to searching through many documents. In particular, issues of multiple users, along with associated security and transaction problems, have not been addressed.

The Neptune system at Tektronix[15] provides a server model for a document database. However, the lowest level of the system is only concerned with the graph structure of the hypertext documents. Other indexing methods, such as keywords, must be done by higher-level applications.

There have been experiments in using traditional, particularly relational, database systems to manage documents. The SODOS project[16] uses a relational database to manage documentation of software systems. Stonebraker et al[17]. have developed a system for representing and editing documents in INGRES. This requires some extensions to the database manager. These techniques do give a "document database", but a specific structure is provided for information. Novel methods of structuring text may be difficult in these systems.

It is interesting to note some of the differences between our database and relational systems. There are a number of advantages of this system for storing documents. Many of these apply to comparisons with Object-Oriented and other general purpose databases, as well to the relational model.

- Pointers are an integral part of the database. Representing the same information in a relational database is difficult (pointers aren't really a part of the document.) In addition, information can be attached to pointers, and they can come in different types.
- We provide for long fields. By recognizing that some fields are more important for searching than others, we can store long fields separately from the rest of the document and accept that these fields will be slower to access. This is known to the user, who can take this into account in designing a database schema.
- This is a simple, special purpose database, providing the minimum functionality needed to handle large numbers of *documents*. The simplicity of the model should allow for a more compact an efficient implementation than a general purpose database.
- The simplicity of the model limits the number of reasonable database schemas for a given application. This will enhance sharing between databases, as related information is likely to be represented in similar ways.

6. Conclusion

We have developed a database manipulation language specifically designed for *documents*. It is capable of expressing complex queries on both individual documents, and *sets* of documents. The structure of information in the database is highly flexible. The structure of a document, and what information can be used in queries, is determined by the application.

This document base operates on the *server-application* principle common to production databases. Issues of security and data integrity can be handled independent of the applications. Emphasis in the applications can concentrate on *presentation* of the information.

This document base will ease the design of hypertext applications. Further research into novel methods of presenting information will be able to build on this system. Researchers in non-technical fields probably have applications for a document database that computer scientists would not envision.

7. Acknowledgements

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

References

1. Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held, "The Design and Implementation of INGRES," *Transactions on Database Systems* **1**(3) pp. 189-222 ACM, (June 1976).

2. M. Astrahan, M. Blasgen, D. Chamberlin, K. Eswaran, J. Gray, P. Griffiths, W. King, R. Lorie, P. McJones, J. Mehl, G. Putzolu, I. Traiger, B. Wade, and V. Watson, "System R: A Relational Approach to Database Management," *Transactions on Database Systems* **1**(2) pp. 97-137 ACM, (June 1976).

3. William Kent, "Limitations of Record-Based Information Models," *Transactions on Database Systems* **4**(1) pp. 107-131 ACM, (March 1979).

4. Thomas W. Malone, Kenneth R. Grant, Kum-Yew Lai, Ramana Rao, and David Rosenblitt, "Semistructured Messages are Surprisingly Useful for Computer-Supported System Coordination," *Transactions on Office Information Systems* **5**(2) pp. 115-131 ACM, (April 1987).

5. D. D. Chamberlin and R. F. Boyce, "SEQUEL: A Structured English Query Language," in *Proceedings of the SIGMOD Workshop on Data Description, Access and Control*, ACM (May 1974).

6. 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 in SIGPLAN Notices **8**(2):II.

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

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

9. 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).

10. 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).

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

12. Randall H. Trigg and Mark Weiser, "TEXTNET: A Network-Based Approach to Text Handling," *Transactions on Office Information Systems* **4**(1) pp. 1-23 ACM, (January 1986).

13. 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 in Sigplan notices **21**(11), November 1986.

14. 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).

15. Norman Delisle and Mayer Schwartz, "Neptune: A Hypertext System for CAD Applications," pp. 132-143 in *Proceedings of the SIGMOD International Conference on Management of Data*, ACM, Washington, DC (May 28-30, 1986).

16. Ellis Horowitz and Ronald C. Williamson, "SODOS: A Software Documentation Support Environment -- Its Definition," *Transactions on Software Engineering* **SE-12**(8) pp. 849-859 IEEE, (August 1986).

17. 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).