Application of OODB and SGML Techniques in Text Database: An Electronic Dictionary System

Jian Zhang

University of Pennsylvania

Department of Computer and Information Science

Philadelphia PA 19104-6389

Email: jian@saul.cis.upenn.edu

Abstract

An electronic dictionary system (EDS) is developed with object-oriented database techniques based on ObjectStore. The EDS is composed of two parts: the Database Building Program (DBP), and the Database Querying Program (DQP). DBP reads in a dictionary encoded in SGML tags, and builds a database composed of a collection of trees which holds dictionary entries, and several lists which contain items of various lexical categories. With text exchangeability introduced by the SGML, DBP is able to accommodate dictionaries of different languages with different structures, after easy modification of a configuration file. The tree model, the Category Lists, and an optimization procedure enables DQP to quickly accomplish complicated queries, including context requirements, via simple SQL-like syntax and straightforward search methods. Results show that compared with relational database, DQP enjoys much higher speed and flexibility. With EDS this paper demonstrates how to apply OODBMS's to systems that handle text information with strong yet varied intrinsic hierarchies.

Key words: object-oriented databases, text database, SGML

1 Introduction

Ever-increasing volumes of documents produced by modern information processing and recording techniques make efficient document access tools urgent necessities. The recognition and marking up of the internal structures of many documents make it possible to increase the efficiency of information retrieval from them, for searching through structures with markup delimiters can be more speedy, reliable, and precise than through a heterogeneous mess of free, unmarked text.

The *Standard Generalized Markup Language* (SGML) is becoming the standard for document description. In SGML, each textual unit, or *element*, is marked with a tag pair, in the form of <..> and </..>. The structure of a document type is standardized through *Document Type Definition* (DTD), which defines for the document type a set of elements and the relationships among the elements [1, 12, 9].

Researchers found that the management of structured documents, such as SGML documents, benefits notably from database support. Marked up data are comparatively easy to load into a database. A number of query languages have been proposed for structured documents [2, 6, 8]. A current trend is to employ *object-oriented database* (OODB) in structured document management. A recent progress on this avenue is a mapping from SGML documents into OODB's with regular OODB query languages extended for handling SGML document retrieval [3].

A typical example of textual documents with strong internal structure is printed dictionaries. Dictionary entries are representable by tree models. Elements of the same type in different dictionaries often bear the same or similar lexical function. This makes dictionary encoding possible. Progress has been made recently in tagging dictionaries with SGML [4, 5]. However, printed dictionaries are among the most complex textual data, unique not only in their use of various typographical conventions but also in the high structural heterogeneity due to differences in language and editorial protocol. Research efforts have been directed to identifying a *generalized dictionary structure* so as to establish a DTD of a *meta-dictionary*. Initial results show that such a DTD could be very complicated [5].

In [3], the OODB classes are defined based on the DTD of the corresponding document. A highly complex DTD, such as that for the document type of the generalized dictionary, will lead to a tree model with a complex maze of classes connected from each other. As opposed to that approach, this paper proposes an OODB-based *electronic dictionary system*, in which all the nodes in the tree model belong to the same class. The node types are indicated only by type symbol. The simplicity is based on two considerations. First, SGML documents are always verified against its DTD after creation, hence the databases built correspondingly are guaranteed to be correct. Second, data queries are often made without knowledge of the data structure, but instead are element-type oriented. The purpose of choosing dictionary as the input data for this study is to show that complex structured document can be managed with OODBMS using simple data models.

Basically, a dictionary is a collection of *entries*, which in turn consist of three principal elements: <fome>, <sense> and <related entry>. Secondary elements, such as <orth>, <pos>, <descrip>, etc., are either contained in the principal elements or may appear anywhere in the entry [9, 12]. While sharing the above basic structure, specific dictionaries have their own features, which can be defined in the EDS configuration file. In this study, the Pequen o Larousse Spanish-Spanish Dictionary is used as a sample [11]. Fig. 1 displays the tree model of the sample dictionary.

EDS was built using the OODB package ObjectStore. ObjectStore provides a tightly integrated language interface to the traditional DBMS features of persistent storage, transaction management, ets., and has strong capabilities in managing large sets of objects, including collections, queries on collections, relation-

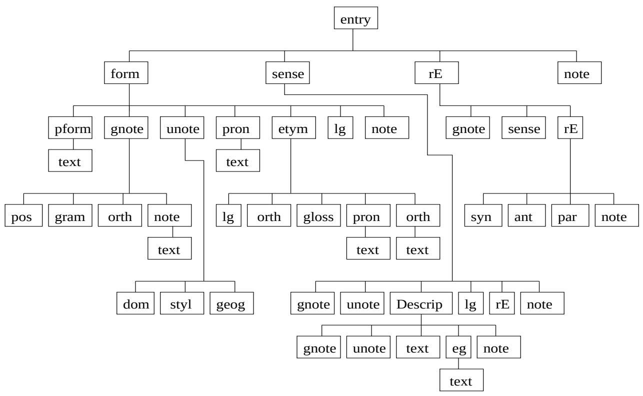


Figure 1: The Tree Representation of the Sample Dictionary Entries

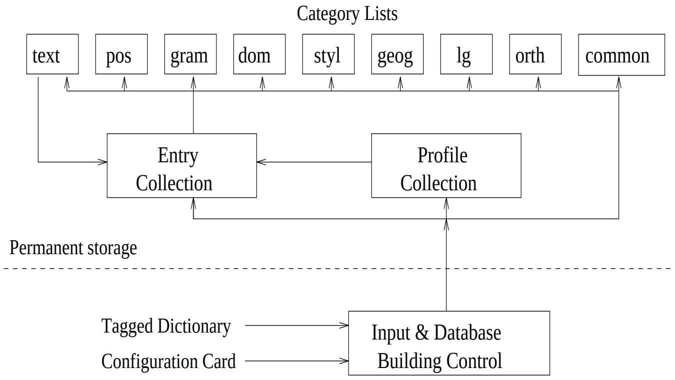


Figure 2: The Architecture of Database Building Program

ships between objects, etc. [10]. All these are important in the development of EDS. Its host language C++ also facilitates the writing of functional modules. This article presents a case study of utilizing ObjectStore to build an object-oriented database using SGML-tagged dictionaries as input data.

This paper is organized as follows. Section 2 and Section 3 are devoted to the description of the two major EDS components: the Database Building Program, and the Database Querying Program. Section 4 shows the unique features of EDS and their usefulness in dictionary querying. Section 5 gives a brief report on the implementation of the system and concludes the paper.

2 The Database Building Program

The EDS contains two basic components: the *Database Building Program* (DBP), and the *Database Querying Program* (DQP). DBP reads in a tagged dictionary and establish the database. Fig.2 displays the architecture of DBP. It is seen that DBP is composed of an input and database-building control module and a permanent database storage. The permanent database storage is divided into three sections, including the *Entry Collection*, the *Category Lists*, and *Profile Collection*.

2.1 The Entry Collection

In EDS, for every entry in the dictionary, a tree is built in the permanent memory, and for every element in the entry, a node is created in the tree. Each node type in the tree corresponds to an element type in the document. Each node is an instance of a node type. To indicate its type, each node contains a *symbol*.

All nodes falls into two categories: either the *branch node*, or the *leaf node*. Branch nodes contain structural information, i.e., the relationship between nodes. Branch nodes have a parent and one or more children, except the *root nodes*, which have no parent. The relationships are arcs between nodes, and in EDS they are implemented as pointers to parent and children nodes. The *inverse data member* facility provided by ObjectStore conveniently handles the data integrity maintaining and automatic updating tasks.

The leaf nodes, on the other hand, contain the textual data of dictionary entries. They have parents but no children. Of nineteen note types in the sample dictionary, eleven are leaf nodes, as are shown in Table 1.

For flexibility in structure representation, all node types are of the same data type called NODE, implemented as a C++ class, with the following attributes:

1. **symbol** – symbol of the node type.
2. **ntext** – text in a leaf node (empty for branch nodes).
3. **parent** – pointer to the parent node (null for root nodes).
4. **children** – a set of pointers to the children nodes (null for leaf nodes).
5. **items** – a list of pointers to items in the related Category List (null for branch nodes).

Explanation on the attribute *items* will be given shortly.

An instance of entry tree is seen in Fig.3(c). The entry has five branch nodes and three leaf nodes. Within each node, the left block contains the type symbol, the middle contains text, and the left may contain pointers to Category Lists.

2.2 The Category Lists

Instead of through 'database roots', as in common OODB's, access to the EDS database is always made through a number of lists called the *Category Lists*. Every words or symbol in an entry belongs to certain lexical categories, such as headwords, pronunciation, part of speech, etc. A Category List contains all the unique values of that lexical category that appear in the dictionary. Normally, many of these values are heavily duplicated in the entries. Therefore, storing the values in the Category Lists rather than in Entry Collection could save space. Moreover, lists are easily indexed, and thus quickly searched. When looking up the dictionary for a lexical items of certain types, instead of checking all the entries, it is only necessary to scan the relevant Category List to find the desired value, and from there to access the related entries via pointers.

Among the Category Lists, the *common* list contains all the commonly-used word in the main language, the *text* list contains the rest of the words in the main language, and the *orth* list holds all the foreign words. The rest of the lists contains items that provide grammatical or usage information (see Table 1 for details). The data type of the cells in all the Category Lists, except the *common* list, is *ITEM*, also implemented as a C++ class, with two attributes:

Leaf Nodes

Symbol	Related Tags	Related Category List	Name	Description
T	_	text	text	Spanish text
N	<syn>	text	synonym	synonym
A	<ant>	text	antonym	antonym
R	<par>	text	paronym	paronym
P	<pos>	pos	pos	part of speech
G	<gram>	gram	grammar	grammartic node
D	<dom>	dom	domain	subject domain
Y	<styl>	styl	style	style usage
O	<geo>	geo	geography	geography origin
L	<lg>	lg	language	language
Z	<orth>/ <gloss>	orth	foreign	foreign text

Branch Nodes

Symbol	Related Tags	Name	Description
E	<entry>	entry	entry
F	<form>	form	morphological form
S	<sense>	sense	meaning
U	<related entry>	related	related entry
H	<pform>	headword	headword
M	<etym>	etymology	etymology
I	<descrip>	descript	description
X	<eg>	example	example

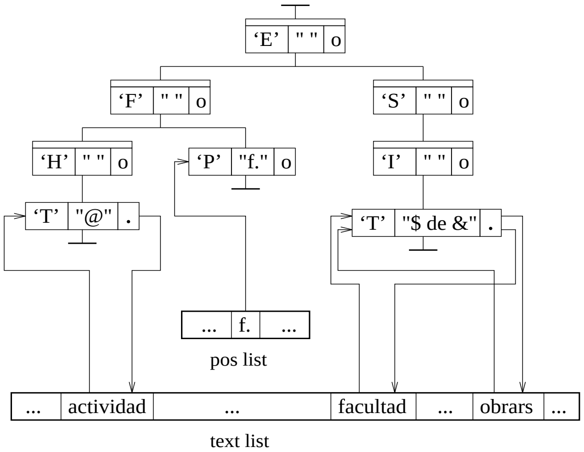
Table 1: Node Types and Related Information for the Sample Dictionary

```
ACTIVIDAD f. Facultad de obrars
```

(a) Original printed form

```
<entry>
  <form>
    <pform> ACTIVIDAD </pform>
    <gnote> f. </gnote> </from>
  <sense>
    <descrip> Facultad de obrars </descrip>
  </sense> </entry>
```

(b) SGML-tagged form



(c) Tree-model form

Figure 3: Representation of an Entry

1. **itext** – text of this list item.
2. **nodes** – a set of pointers pointing to entry tree nodes that refer to this list item.

The pointers in *nodes* point to those leaf nodes whose *ntext* attribute contains the *itext* of the item, i.e., *ntext* is a superstring of *itext*.

The attribute *nodes* of class ITEM and the attribute *items* of class NODE represent the relationship between entry tree nodes and Category List items. Since every leaf node always belongs to a lexical category, every leaf node has a related Category List. On the other hand, every Category List can be related to multiple nodes. The relationship is thus one-to-many.

The relationship between nodes and items can be utilized not only for querying purpose, but also for space saving. Duplications of list item values in entry nodes can be eliminated by replacing such a string in *ntext* of a node with a special sign, and at the same time inserting to *items* of the node a pointer pointing to the item in its related Category List whose *itext* attribute equals the string. Considering the storage space of pointers, it is wise to replacing a string *s* with a sign only it is long enough. A replacement rule could be:

```
If length(s) > size_of(pointer) + size_of(sign)
  then do replacement
else
  copy s into ntext.
```

It is seen in Fig.3(c) that in the text node containing three words, only the two longer words are replaced by signs, and the short word 'de' remains in *ntext*. Different replacement signs are used to indicate different case patterns of the original words.

The separation of commonly-used words from the rest of the vocabulary is also aimed to save space. If commonly- used words are included in the *text* list, then their *nodes* attribute could take up a large amount of space, for they are expected to appear in the text of many entries. Therefore, the *common* list is just a string array. As a result, a query based solely on common words requires combing through the entire Entry Collection, which is inefficient. Fortunately, such queries are rare. Commonly-used words are specified in the configuration file.

2.3 The Profile Collection

A node is an instance of a node type. Each node type has a set of attributes, including its symbol, the related tag, the related Category List, its name in query, etc. The attribute values of node types are stored in the *Profile Collection*. To store these information, a PROF class is defined, with the attributes of:

- **symbol** – symbol of the node type.
- **tag** – start-tag for the corresponding entry element.
- **list_id** – ID of related Category List.
- **name** – name used in queries to refer to the node type.

The configuration file has a profile section. Each row in the section contains all the attribute values for a node type, in the order

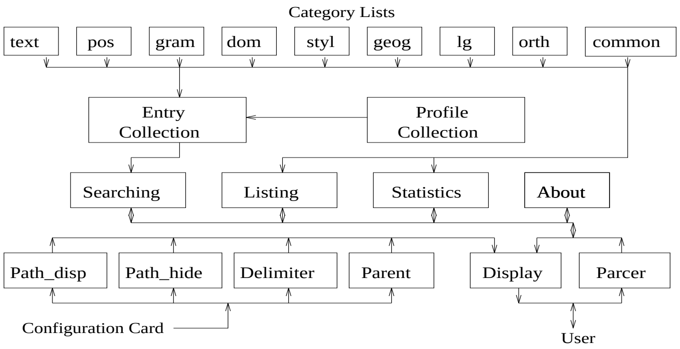


Figure 4: The Architecture of Database Querying Program

of tag, ID of related Category List, node symbol, and name used in query, for instance:

<sense>	0	S	sense
<related entry>	1	U	related
...			

In the above example, the node type named *sense* has the start-tag '<sense>', and a symbol 'S'. Being of branch node type, it has no related Category List. On the other hand, <related entry> is a leaf node type related to the *text* list, the first Category List.

3 The Database Querying Program

DQP is another major component of EDS. DQP receives queries from the user and displays the results of the querying. Fig.4 shows the architecture of DQP. Based on the permanent database storage, the DQP provides two principle querying functions: entry searching and item listing. A minor function is statistics reporting, which prints the general information on the dictionary database. The system interacts with the user through the parser module to get input and through the display module to deliver output. DQP provides various displaying formats especially the path-display. To speed up query processing, an optimization procedure is adopted.

3.1 Entry Searching

Receiving from the user a set of criterion, DQP searches the database for entries that satisfy the criteria. The syntax of an entry search query is:

```
select <list of node type names | all | path>
           [formatted]
where  <leaf node type name | any> = <value>
           [,<leaf node name> = <value>] [, ...]
         [in <ancestor node type name>]
         [and ...]
        [or ...]
```

The syntax is SQL-like, although the 'from' statement is missing since there is only one source — the entry collection.

The optional 'in' statement forms the context requirement, which specifies within which element type (the ancestor) the matched element should be. Examples will be given in next section. Both 'and' and ',' represent intersect operation, but 'and' combines multiple criterion clauses, while ',' combines multiple criterion values within a clause.

Wild-card expressions can be used in several places. One may use 'all' or 'path' in place of node type name after 'select', use 'any' in place of leaf node type name after 'where', or put *'s anywhere in a string value. The concept of 'path' will be explained later. The meaning of the other expressions are obvious.

The optional keyword 'formatted' specifies that the display of matched entries should be formatted, including embracing important lexical items (headword, part of speech, etc.) with specified parenthesis, adding number indexes to senses, adding alphabetic indexes to descriptions, and making proper indentations. Formating information is also given in the configuration file.

3.2 Item Listing

This function selects from a Category List all the items that equal a user-specified value. The syntax is:

```
list <name of node type | all> = <value> [/f].
```

The 'f' switch specifies the display of the frequency of the items. Item listing helps the user choose criterion values while formulating a query. With the query pick function in ObjectStore, the implementation of the listing is straightforward.

4 Unique Features of EDS

4.1 Element-type-orientation

Due to the high heterogeneity of structures in dictionary entries, a search criterion is often applied to a specific element type, regardless the position of the matched elements in entries. In relational databases (RDB) queries, however, element positions are important, since the positions determine in which tables these elements will appear as attributes. In the following example, the language element <lg>, is part of the form element <form> in the first entry, is within an etymology element <etym> in the second, and belongs to a sense element <sense> in the third. A RDB query based on <lg> must specify all the possible positions of <lg> and inevitably become very clumsy.

```
<entry> <form> <pform> AD HOC </pform> ...
  <lg> lat. </lg> </form>
  <sense> ... </sense> </entry>

<entry> <form> <pform> TIENDA </pform> ...
  <etym> ... <lg> lat. </lg> ... </etym> </form>
  <sense> </sense> ... </entry>

<entry> <form> <pform> ACAMALAR </pform> ...</form>
  <sense> ... <lg> lunf. </lg> ... </sense>
  <sense> ... </sense> </entry>
```

On the other hand, with the Category Lists and tree structure, EDS easily supports this element-type-orientation. Receiving a query as simple as

```
select all where language = lat.
```

DQP will go to the *lg* list, find the item *lat.*, and follow pointers there to all the entries which have a language node with the value *lat*.

4.2 Context Requirements

Some queries may use constraints on the relative position of matched elements. For instance, the user is interested in a <lg> that must covers the whole entry, i.e., it should be within <form>. To express the context requirement, one only needs to add the 'in' statement to the original query

```
select all where language = lat. in form
```

After reaching all the matched leaf nodes, DQP will check the required ancestry using the parent pointers. In the above example, only the first two entries will be picked up. It is imaginable that in a RDB various ad hoc algorithms have to be employed to fulfill the task.

4.3 Path Display

It is common that some dictionary entries are of considerable size, containing dozens of related entries and senses. A query, however, is often aimed to look at a few pertaining elements. Displaying the entire entry, which could takes several screens, is distracting. What need to be displayed here are those key elements which roughly form a *path* from the headword to the matched leaf node. Such a path has the feature that when standing alone, it becomes an independent entry. For example, given a lengthy entry:

```
[ALGOD'ON]
  (m.)
    1. Planta malv{'a}cea, cuyo fruto contiene de
       quince a veinte semillas envueltas en una
       borra muy larga y blanca: el algod{'o}n es
       originario de la India.
    2. ......
    ......
    5. (Fig.) y (fam.)
       Criado entre algodones, criado con mimo y
       delicadeza.
  --   El algod{'o}n es originario de la India.
       ......
```

With path displaying, a query targeting the style element *Fig.* returns

```
[ALGOD'ON]
  (m.)
  (Fig.) y (fam.)
    Criado entre algodones, criado con mimo y
    delicadeza.
```

which is clear, concise, and to the point. Considering the fact that a query could capture a large number of entries, the usefulness of path become quite evident. The concept of path is somewhat analogical to the concept of view in relational database operations. While views are created during system-user interaction, paths are defined in the configuration file. Paths are leaf-node-specific, i.e., each leaf node type has its own path, which are specified in the configuration file.

4.4 Frequency Counting

In linguistic researches, frequency of usage of lexical items is often an important parameter. In EDS, frequency data can be easily obtained, such as the 'f' switch in item listing. Given a lexical item, the number of times it appears in the dictionary roughly equals the cardinality of the pointer set in the *items* attribute of that list item. Context constraints can be readily added through the 'in' statement. Although frequency constraint is currently not included the query, it can be added with no difficulty.

4.5 Optimization

For a single-criterion search, which is based on only one item value of certain lexical category, EDS scans that Category List to reach the given item and from there to reach all the leaf nodes that contain the item, and finally, through the parent-children relationship to get to the root nodes of the entries which are sought for. For a search on two criteria, however, there are two alternative approaches:

1. **Intersecting** – use the above method to get two sets of matched entries s_1 and s_2, and intersect the two sets to obtain the final set s, i.e., $s = s_1 \cap s_2$.
2. **Filtering** – choose one criterion c_1 to obtain s_1, and filter s_1 with another criterion c_2 to delete unmatched entries. The result can be expressed as $s = filter(s_1, c_2)$.

Small matched sets and large entry sizes favor intersecting, while large matched sets and small entry sizes justify filtering. Formally, given s_1 and s_2, we choose intersecting if

$$|s_1| \cdot |s_2| \cdot t_u < |s_1| \cdot n_1 \cdot t_f$$

i.e.

$$|s_2| \cdot t_u < n_1 \cdot t_f$$

where $|s_i|$ is the cardinality of set s_i, n_i is the average number of nodes in the entries in s_i, t_u is the time for checking the equality of a pair of elements, and t_f is the time for traversing one arc and comparing two text nodes. Note $|s|$ can be estimated from the cardinality of the set attribute nodes in the criterion item, with the assumption that the duplication rate for any list item in an entry is close to one. Meanwhile, n_i can be estimated by the average number of nodes for all the entries in the dictionary.

Based on the above analysis, an *optimization procedure* for entry search on multiple criteria can be defined as follows:

1. For each search criterion c_i, $i \in I$, get the matched set s_i;
2. Let $s_1 = \min_{|S_i|, i \in I} s_i$ and $I = I - i$;
3. If $I = \emptyset$, stop, else let $s_2 = \min_{|S_j|, j \in I} sj$; let $s_1 = s_1 \cap s_2$ if $|s_2| \cdot t_u$¡$n_1 \cdot t_f$, otherwise let $s_1 = filter(s_1, c_2)$ and $I = I - j$;
4. if intersecting is used in Step 3, go to Step 3; otherwise (filtering is used) go to Step 5;
5. if $I = \emptyset$, stop, else let $s_1 = filter(s_1, c_j)$, $j \in I$; let $I = I - j$; go to Step 5;

It is seen that since s_2 is chosen in the ascending order of set cardinality, once filtering is justified, it will be used for all remaining sets.

It should be noted that this procedure is applied for criteria linked by logic **and**'s. For logic **or**'s, the union operation is always used.

5 Test and Conclusions

The intrinsic tree model of SGML makes it a suitable field for OODB applications. Coupling OODB techniques with with SGML, EDS has a number of advantages over traditional relational databases, including:

- High flexibility in accommodating lexical and structural variations across different dictionaries. The only modifications to be made for a new dictionary are in the configuration file.
- Simplicity in implementing query searching. Searching is based on element types, instead of on their position. The latter is typical in a RDB yet is impractical when facing the high structural heterogeneity among dictionary entries. Context condition checking is easily done in EDS.
- High speed. The employment of Category Lists, the tree structure of entries, and the extensive use of object pointers greatly expedite the search process. Moreover, an optimization procedure raises the speed still further by making intelligent choices between alternative approaches.
- Flexibility in displaying query results. The special path format enables smart display by showing only those elements that are relevant to the search purpose, thus avoiding clumsy and distracting display of lengthy entries.

An experimental database, named SEDS (the first 'S' stands for Spanish), was built. Because of space limitation, only a small portion (about 1/12) of the sample dictionary are loaded into the database. The building of SEDS takes approximately 20 minutes on a SUN workstation. Table 2 lists the statistics of SEDS. Of the eight Category Lists, only sizes of the *text* list and the *orth* list are presented. All the other lists has fewer than 40 items. The table reveals several facts:

First, it can be calculated that the average word length is above five (150715 / 27720 5.44). Therefore, according to the replacement rule, space can be reduced for about half the words, which is substantial.

Second, compared with the input text file, the database takes more space (almost fourfold). The increase in space may stem from the following reasons:

1. duplication of words in the Entry Collection and the Category Lists;
2. implementation of entry structure, i.e., spaces for nodes in entry trees;
3. implementation of node-item relationships, i.e., space for pointers between entry nodes and list items;
4. overhead needed for management of ObjectStore collections.

While (1) and (4) seem unchangeable, space reduction can be realized for the other two through:

- shifting words from the text list to common list, which decreases the number of item-to-node pointers.
- using item-to-root pointers rather than item-to-node pointers. This reduces the number of pointers, and makes the parent attribute in a node unnecessary.
- implementing different data structures for the branch node and the leaf node, so as to eliminate the unnecessary ntext, items, or children attributes.

It should be pointed out that reduction in space often means slowdown in speed. So the validity of the suggestions needs more tests.

Size of tagged file	544,432
Number of entries	1,462
Number of words	27,720
Number of characters	150,715
Length of text list	7,191
Length of orth list	283
Size of database	2,490,368

Table 2: Key Statistics of SEDS (sizes in bytes)

With the current architecture and database, the query execution by the DQP is extremely fast, and the delay is unnoticeable. On the other hand, an existing relational database on the same dictionary (the whole dictionary) requires more than ten seconds to complete similar queries, with much less flexibility in choosing lexical categories and context requirements. A more accurate comparison of speed, however, must wait until the whole dictionary is loaded into the SEDS database when enough space is available.

Acknowledgements: I am grateful to Dr. P. Buneman and Dr. Limsoon Wong for their help in the writing of the paper. I also thank the Language Analysis Center, University of Pennsylvania for their provision of dictionary data.

References

[1] Barron, D. (1989) "Why use SGML?", *Electronic Publishing*, Vol. 2(1), pp. 3-24.

[2] AIS Berger-Levrault (1993) "SGML/Search, Description du Langage". Internal Document, 34 Avenue du Roule 92200 Neuilly sur Seine.

[3] Christophides, S., S. Abiteboul, S. Cluet and M. Scholl (1994) "From Structured Documents to Novel Query Facilities", *ACM SIGMOD 1994 Conference on Manager of Data*.

[4] Fought, J., M. Wesler, and H. Daverport (1992) "Extending SGML concurrent structures: Toward computer-readable meta-dictionaries", working paper.

[5] Fought, J. and C. Van Ess-Dykema, (1990) "Toward an SGML Document Type Definition for Bilingual Dictionaries", Document Number: TEI AIW20. Text Encoding Initiative: Chicago and Oxford.

[6] Gonnet, G. H. and F. W. Tompa (1987) "Mind You Grammar: a New Approach to Modeling Text", in *VLDB'87* Brighton, pp. 339-346

[7] Kim, Won, (1990) *Introduction to Object-Oriented Databases*, MIT Press, Cambridge, Mass.

[8] Macleod, I. A. (1990) "Storage and Retrieval of Structured Documents", *Information Processing & Management*, 26:197-208.

[9] Martin Bryan, (1988) *SGML an Author's Guide to the Standard Generalized Markup Language*, Addison-Wesley Publishing Company.

[10] Object Design Inc. *The ObjectStore User's Manual.*

[11] *Pequen o Larousse Ilustrado*, Edited by Ramon Garcia-Pelayo y Gross. Ediciones Larousse: Barcelona.

[12] SoftQuad Inc., *The SGML Primer*.