Integrating a Structured-Text Retrieval System with an Object-Oriented Database System

Tak W. Yan

Department of Computer Science
Stanford University
Stanford, CA 94305

Jurgen Annevelink

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304

Abstract

We describe the integration of a structured-text retrieval system (TextMachine) into an object-oriented database system (OpenODB). Our approach is a light-weight one, using the external function capability of the database system to encapsulate the text retrieval system as an external information source. Yet, we are able to provide a tight integration in the query language and processing; the user can access the text retrieval system using a standard database query language. The efficient and effective retrieval of structured text performed by the text retrieval system is seamlessly combined with the rich modeling and general-purpose querying capabilities of the database system, resulting in an integrated system with querying power beyond those of the underlying systems. The integrated system also provides uniform access to textual data in the text retrieval system and structured data in the database system, thereby achieving information fusion. We discuss the design and implementation of our prototype system, and address issues such as the proper framework for external integration, the modeling of complex categorization and structure hierarchies of documents (under automatic document schema importation), and techniques to reduce the performance overhead of accessing an external source.

1 Introduction

With the advent of document structure markup standards such as SGML [8] and ODA [3], structured documents have recently received a lot of attention. The logical structure of text constitutes important information for querying and/or retrieval, but traditional text retrieval systems typically do not make full use of the document structure. New research proposals and prototypes, such as [4,11,14], and commercial text retrieval systems, such as [1], have emerged to address this problem.

Structured documents also lend themselves to be managed by database systems. A lot of efforts have been made to add text retrieval capability to both relational and object-oriented systems. Past efforts to model structured text in a relational system have met with difficulties [15], and object-oriented database systems have been developed partly with the aim to store and manage text and other complex data. In either case, the database system stores the documents and is extended with text indexing and retrieval capabilities.

In this paper, we present a more light-weight, flexible approach of integrating a text retrieval system (TextMachine [1]) and an object-oriented database system (OpenODB [9]). The text retrieval system retrieves structured documents. It exists alongside with the database system. Documents are stored in, and all indexing and retrieval of documents are performed by, the text retrieval system. The object-oriented database system treats the text retrieval system as an external information source. It communicates with the text retrieval system through the latter's application programming interface, submitting queries and receiving answers. The interface is encapsulated by the so-called external function capability of the object-oriented system, and the underlying processing is transparent to the user; i.e., the user is unaware that the text retrieval system is an external source.

Such integration seamlessly combines the best of both worlds – the efficient and effective indexing and retrieval of structured documents performed by the text retrieval system, and the rich modeling and general-purpose querying capabilities supported by the database system – resulting in an integrated system with querying power beyond those of the underlying systems. The integrated system also provides uniform access, i.e., the ability to uniformly query over both the textual data managed by the text retrieval system and the structured data stored in the database system, thereby achieving information fusion.

To illustrate our claims above, let us consider some sample queries that can be processed by the integrated system. We describe the queries in English here, before we introduce the query languages and models of the two systems. (The actual queries will be presented later in the paper.) Suppose we have a collection of medical progress notes residing in the text retrieval system. Also suppose we have some structured information, including names and specialties, about some physicians stored in the database system. Now consider a user who is interested in retrieving paragraphs in the progress notes that mention "bronchitis" and also the name of a physician whose specialty is "respiratory." The user submits a query, in the database query language, to the integrated system (the actual query is (Q3) in Section 3.3.3). When the query is processed, the names of the physicians with specialty "respiratory" are selected from the database and each name becomes part of a query to the text retrieval system. A number of queries (equal to the number of names selected) are then submitted to the text retrieval system and the combined results are returned to the user. This way, the user transparently makes use of structured data to query the text retrieval system. Furthermore, the text data retrieved can then be extracted and assimilated into the structured database.

In the next example, we demonstrate that the retrieval power of the integrated system is beyond that provided by the underlying systems. The retrieval model of TextMachine is similar to the tree inclusion primitive, presented recently in [11]. The essential idea of the primitive is to represent documents and queries as trees; a query specifies a partial tree pattern that the retrieved document trees must contain. (We will further discuss this in Section 2.2.2.) In the integrated system, we are able to retrieve text data using queries not expressible with the tree inclusion primitive. Query (Q4) in Section 3.3.4 is such an example. The user submitting this query uses two partial patterns to select paragraphs from the text retrieval system. He further adds the constraint that a paragraph retrieved by the first pattern must be adjacent to a paragraph retrieved by the second pattern, using database query language constructs. We cannot express this adjacency constraint in TextMachine or using the tree inclusion primitive alone.

In this paper, we describe the design and implementation of our prototype integrated system. We address the following issues:

- Framework for external integration

  We investigate the use of the database system's external function capability to access the retrieval system. We show what abstraction of the external source is desirable for integration and how document schemas from the text retrieval system are imported. We show how, in spite of the nature of the external integration, we achieve a tight coupling of query language through query translation.

- Modeling text

  We discuss the use of object-oriented concepts – objects, types, and functions – to model complex categorization and structure hierarchies of documents. We address the modeling issues involved when we import document schema from the text retrieval system into the integrated system.

- Performance issues

  A potential drawback of this approach is the performance penalty of accessing an external source. However, we can minimize the overhead in a number of ways, such as efficient techniques to perform conversion between keys from external sources and database object identifiers (OIDs), optimizing the processing of external functions, and so on. In our prototype, we investigate the use of algorithmic OID conversion.

Although the work reported in this paper is on integrating with a particular structured-text retrieval system, the approach and techniques are *orthogonal* to the underlying retrieval model and language, and can be extended to other text retrieval systems as well. In fact, we have integrated OpenODB with another system, Wide Area Information Servers (WAIS) [10], which is a networked information system. This demonstrates the generality of our techniques.

The rest of the paper is organized as follows. In Section 2 we describe the testbed systems of our integration. In Section 3 we present the integrated system. The implementation details of the prototype are covered in Section 4. We survey related work in Section 5. Finally, Section 6 is for conclusion and discussion of future work.

2 The Testbed

In this section, we briefly describe the object-oriented database system and the text retrieval system that we have integrated. The scope of our paper limits this to a short description of relevant topics. The reader is referred to [1] and [9] for detailed coverage of the systems.

2.1 OpenODB: a Functional Object-Oriented Database

OpenODB is an object-oriented database system developed by Hewlett-Packard. OSQL [2] is a database programming language for OpenODB. It combines the object-oriented features found in such languages as C++ and Smalltalk with a query capability that is a superset of the familiar SQL relational query language.

The OSQL language is centered around three basic concepts: objects, types, and functions. *Objects* represent the real-world entities and concepts from the application domain that the database is storing information about. Each object has a unique object identifier (OID). *Types* are used to classify objects on the basis of shared properties and/or behavior. Types are also used to define the signature of functions, i.e., their argument and result types. Types are related into a subtype/supertype hierarchy that supports multiple inheritance. The type hierarchy enforces type containment, i.e., if an object is an instance of a given type T, it must also be an instance of all supertypes of the type T. OSQL surrogate objects can be instances of any number of types, even if the types are not related by a subtype/supertype relationship.

Functions are used to model attributes of the object, interobject relationships, and arbitrary computations. One of the key distinctions of OSQL as compared to other models, e.g, those inspired by various programming languages, is this unifying notion of a function to model stored and derived attributes, stored and derived relationships and arbitrary computations (behavior). An OSQL function takes one or more objects as arguments and may return an object as a result. OSQL functions can be overloaded, i.e., there can be multiple functions with the same name but different argument types. Functions have extensions. The extension of a function is the mapping from its argument(s) to its results. Function extensions can be explicitly stored, or they can be computed. Functions whose extension is computed can be implemented either as an OSQL expression, or as a program (subroutine, procedure) written in a general purpose programming language (e.g., C). These latter are called external functions and give OSQL a unique form of extensibility by allowing the encapsulation of (entry points in) external libraries, to access information, data, and functionality outside of an OpenODB database. We made extensive use of external functions in our integration.

OSQL supports a query language whose semantics is based on domain calculus. The OSQL select function provides the basic query facilities of OSQL and closely resembles the select statement of SQL. For example, suppose we have a type Physician with an attribute specialty in our database. Then the query

```
select p for each Physician p where specialty(p) = 'respiratory';
```

returns the OIDs of the physicians whose specialty is "respiratory."

⟨DOC⟩⟨DOCTITLE⟩Integrating a Structured-Text Retrieval System with an Object-Oriented Database System⟨/DOCTITLE⟩ .... ⟨SECT⟩⟨SECTTITLE⟩Introduction⟨/SECTTITLE⟩With the advent of document structure markup standards such as .... Finally, Section 6 is for conclusion and discussion of future work.⟨/SECT⟩

Figure 1: A sample excerpt from a TM-tagged document

2.2 TextMachine

TextMachine (hereafter abbreviated to TM) is a structured-text retrieval system from Alliance Technologies [1].

2.2.1 Structured Documents

TM provides a text preparation system that marks up raw ASCII documents with tags that identify the various structural *components* of documents, such as chapters, sections, paragraphs, and sentences. It also identifies domain-specific information, called *attributes*, in documents; e.g., proper names, drug names, or phone numbers. The particular components and attributes identified for a certain kind of documents are user-defined. We do not go into the details of the mark-up process in this paper. Figure 1 shows a sample excerpt from a TM-tagged document.

Documents are grouped into *databases*. Documents in the same database share the same logical structure: they are tagged by the same mark-up procedure. The logical structure of documents in a database is described by some sort of document schema, called the *database definition file* (*DDF*). The DDF specifies (the tags of) the components and their nesting relationship, the titles of certain components (if defined), and the attributes. Different databases may have different structures, described by different DDFs. This is somewhat analogous to the Document Type Definitions (DTD) in SGML. Figure 2 shows the relevant information from the DDFs of two TM databases. Figure 2A is for a technical memo database (of which the document in Figure 1 is a sample document) and Figure 2B is for a medical progress note database.

2.2.2 Retrieval Model and Query Language

The retrieval model and query language of TM bear resemblance to recent proposals in the literature [4,11,14], and are closest to the tree inclusion primitive [11]. Here, we briefly discuss the primitive, as formalized in [11].

The tree inclusion primitive applies to general tree structures. If p and t are trees, an *embedding* of p in t is an injective function e from the nodes of p to the nodes of t. An embedding e *preserves* a binary property ϕ between nodes, if for any pair of nodes u and v of p, we have $\phi(u, v)$ holds in p if and only if $\phi(e(u), e(v))$ holds in t. Given a set S of properties to be preserved, an S-*embedding*

level0	DOC
level1	SECT
level2	SUBSECT
title0	DOCTITLE
title1	SECTTITLE
attribute	PNAME

A

level0	DOC
level1	SECT
level2	PAR
level3	SENT
title0	DOCTITLE
title1	SECTTITLE
attribute	DRUG

B

Figure 2: Sample DDFs for two TM databases

is an embedding that preserves the properties of S. Given a pattern tree p, a target tree t, and a set S of properties, p is an *included tree* of t if there exists an S-embedding of p in t.

In TM, we may view documents as labeled trees. The height of a document tree is equal to the number of levels in the DDF plus one. The root represents the top level component in the DDF, its children represent the second level components, and so on. The leaves represent the individual words. The interior nodes are labeled by the level names, while the leaves are labeled by the words. As an example, the right-hand-side of Figure 3 is the tree of a document instance described by the DDF in Figure 2B.

In the terminology of [11], TM retrieval preserves labeling and ancestorship properties, and is unordered (i.e., does not preserve left-to-right ordering). In Figure 3, the partial pattern on the left is an included tree of the document tree on the right under the preservation of these properties. To retrieve documents, the user specifies a partial pattern p of the desired document trees. All documents that satisfy the tree inclusion primitive are returned (i.e., all documents of which p is an included tree). [1]

The TM query language supports different retrieval operators for databases described by different DDFs. For example, for a database described by the DDF in Figure 2B, we have the following "window operators": W/DOC, W/SECT, W/PAR, and W/SENT. An operator takes an arbitrary number of words as arguments and we can also compose the operators to specify complex partial patterns. For example, the query

[1]Note that the unit of retrieval in TM is a document. We believe that it is desirable for the user to retrieve the minimal document components that satisfy a specified pattern; e.g., the user may want to retrieve at the paragraph level, instead of entire documents. Thus, in our integration, we allow the minimal components to be retrieved.

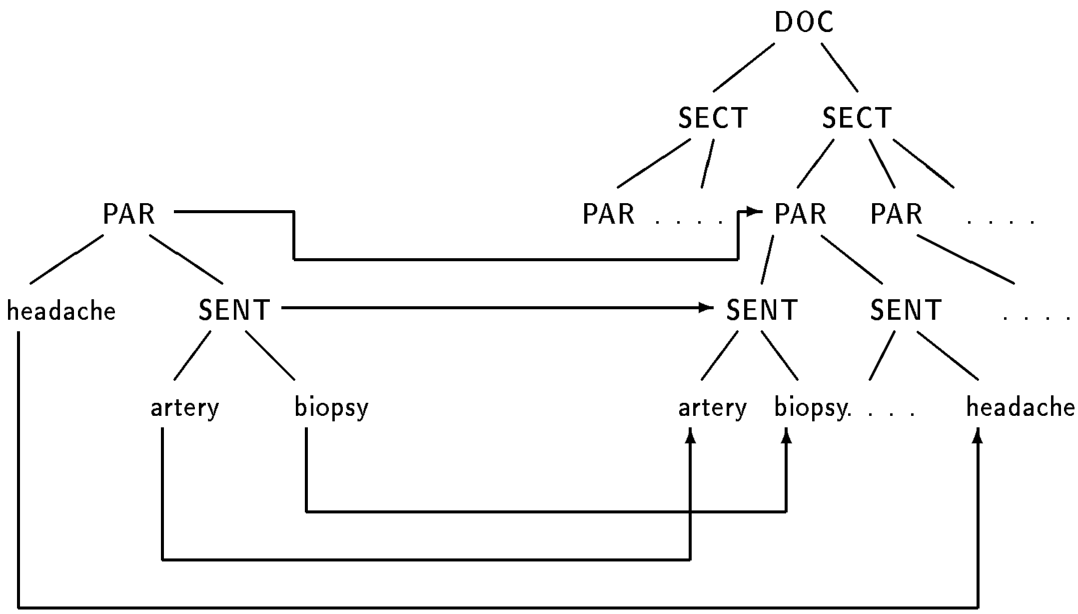

Figure 3: Tree inclusion

W/PAR[headache (W/SENT[artery biopsy])]

specifies the pattern shown on the left-hand-side of Figure 3. It retrieves documents that have a paragraph containing the word "headache" and a sentence with the words "artery" and "biopsy."

TM also provides the ability to search on component titles and attributes. For example, the query

W/SECT⟨course⟩[headache DRUG[tylenol]]

is for sections with the word "course" in the title, and containing the word "headache" and the drug name "tylenol." In addition to these DDF-specific operators, TM supports the OR operator and the proximity operator W/n (n an integer). Extensions such as phrases and truncation are also provided.

3 The Integrated System

We present the integrated system according to the three object-oriented concepts: objects, types, and functions.

3.1 The Use of Objects

In our integration, we provide a view of text objects, which are used to model the individual structural components of documents. The granularities of text objects for a given kind of document

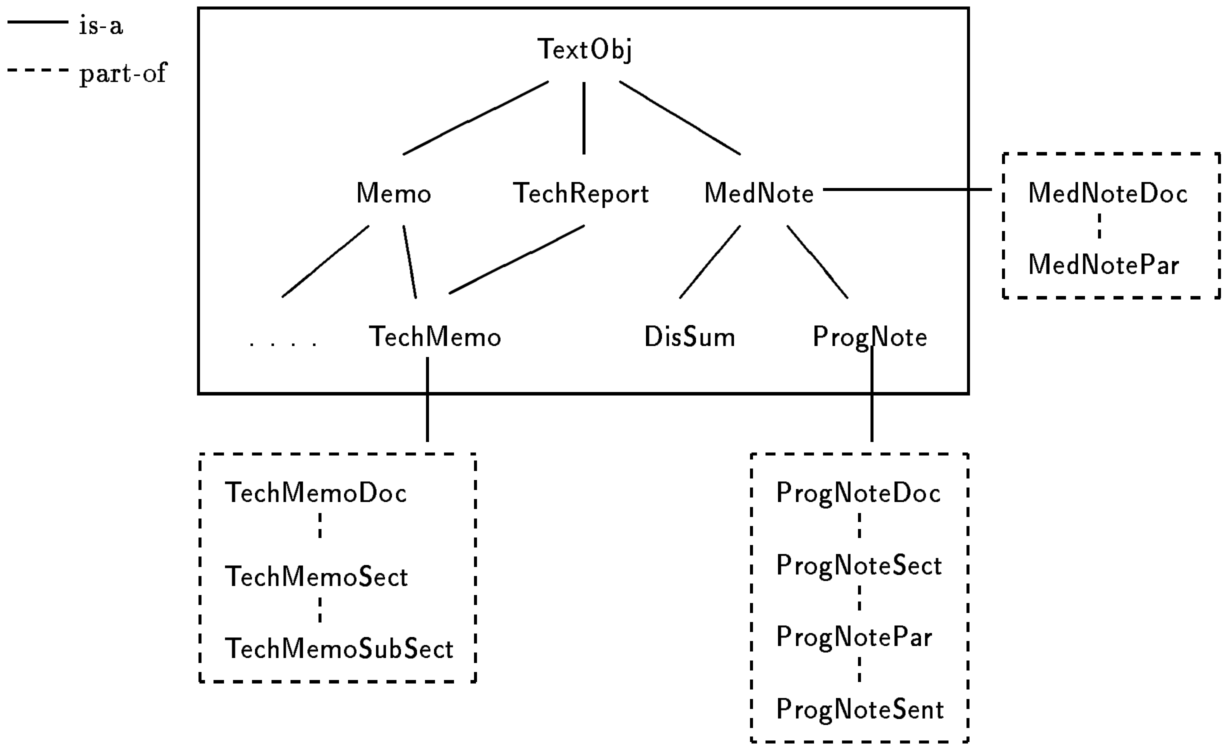

Figure 4: Text categorization (is-a) and structure (part-of) hierarchies for a database instance

are as defined by the associated document schema (DDF). For example, in a progress note document defined by the schema shown in Figure 2B, we have text objects that model documents, sections, paragraphs, and sentences. Each text object is uniquely identified by an OID.

3.2 The Use of Types

TM partitions documents into databases, and queries can be limited to a number of selected databases. However, this two-level (database-document) hierarchy is not sufficient to model more complex relationship of document types. In the integrated system, we model text by two kinds of hierarchies, which are orthogonal to each other. The first, called the *categorization hierarchy*, models the relationship between the different kinds or categories of texts (e.g., medical notes, technical reports). The second, called the *structure hierarchy*, models the relationship between the structural components within a certain category of text. Below we first describe each kind of hierarchies in turn, then we explain the interplay between them, and show how the hierarchies are automatically constructed from meta information extracted from the DDFs.

3.2.1 Text Categorization Hierarchy

The text categorization hierarchy is an *is-a* hierarchy that defines the relationship among different kinds of text. The system-defined type TextObj is the supertype of all text objects. In any database instance of the integrated system, we have a single text categorization hierarchy. Different database instances may have different categorization hierarchies. For example, Figure 4 shows the categorization hierarchy (in solid lines) of a particular database instance. The subtypes of TextObj include types Memo, TechReport, and MedNote. Type MedNote in turn is the supertype of types ProgNote (for progress notes) and DisSum (for discharge summaries). Note that OpenODB allows multiple supertypes; e.g., type TechMemo is subtype of both types Memo and TechReport.

3.2.2 Text Structure Hierarchy

Within a text category, we have text objects that correspond to different structural components. These text objects are organized into a *part-of* hierarchy, or structure hierarchy. In a database instance, there are multiple structure hierarchies, one for each node in the categorization hierarchy. For example, in Figure 4, we show the structure hierarchies for three of the nodes in the categorization hierarchy. For text objects of type ProgNote, we have type ProgNoteDoc for objects that correspond to entire progress note documents, ProgNoteSect for sections within progress notes documents, ProgNotePar, and ProgNoteSent. A ProgNoteSect object is part of a ProgNoteDoc object, and a ProgNotePar object is part of a ProgNoteSect object (and also part of a ProgNoteDoc object). Structure hierarchies are effected by functions that allow the user to traverse from one component to another (described in Section 3.3.2).

3.2.3 Interplay between the Hierarchies

As just mentioned, each node in the categorization hierarchy has its own structure hierarchy. A leaf node corresponds to a TM database and its structure hierarchy directly maps from the document schema associated with the TM database. For an interior categorization node, its structure hierarchy must be explicitly constructed by the user. (Automatic generalization from the leaf nodes would lead to semantic difficulties; see last paragraph in Section 3.2.4.) The structure types are made into subtypes of the categorization type. For example, in Figure 4, ProgNoteDoc, ProgNoteSect, ProgNotePar, and ProgNoteSent are all subtypes of ProgNote. A ProgNotePar object is a ProgNote object, a MedNote object, and a TextObj object. Note that the categorization types are all abstract types; i.e., every ProgNote text object also belongs to a certain structure subtype, such as ProgNotePar.

We may contrast this with an alternative. Suppose that in our example, we create instead the structure hierarchy Doc – Sect – Par – Sent and make every text object belong to two immediate types: a structure type and a categorization type. That is, a ProgNotePar object in the original scheme would have two immediate types in the alternative scheme: ProgNote and Par. This has some drawbacks. First, different TM databases may have identical names for components that are related by different part-of relationships. For example, in a magazine database we may have a Section component being part of an Article, in another we may have the opposite. The first scheme handles this situation easily, but complications arise for the alternative. Another drawback is that the user may want to query on or to have a function for a specific categorization+structure type; e.g., a function for displaying progress note sections. This is difficult to achieve using the second scheme.

3.2.4 Importation and Generation of Hierarchies

These hierarchies are automatically constructed as we *import* TM databases into OpenODB as leaf subtypes in the categorization hierarchy. While the structure hierarchy directly maps from the TM document schema, the information for building the text categorization hierarchy has to be explicitly specified during importation. Essentially, we have to name the immediate categorization supertypes of the type that we are importing.

As an example, the type ProgNote is imported as follows:

```
import /users/tyan/ProgNote/db/ProgNote MedNote
```

Import is a C program that generates the OSQL statements for building the hierarchies. The first argument to import is the path of the TM text database. The name of the database (ProgNote in the example) is used as the name of the text categorization type. The first argument is followed by one or more names of the immediate supertypes of the imported type. In our example, ProgNote is imported as a subtype of MedNote, which we assume has already been created in the database (as a subtype of TextObj). The import program reads the DDF and outputs the following OSQL statements:

```
create type ProgNote subtype of MedNote;
create type ProgNoteDoc subtype of ProgNote;
create type ProgNoteSec subtype of ProgNote;
create type ProgNotePar subtype of ProgNote;
create type ProgNoteSent subtype of ProgNote;
```

The importation also creates OSQL statements that insert information into the OpenODB database needed for query translation and algorithmic OID generation, discussed later on. Once the importation is done and the user loads the statements into a database of the integrated system, he can start querying TM using functions described in the next section.

Note that in importation, no automatic generation of structural subtypes is done for the categorization supertypes. As an example, in importing the ProgNote subtype, the import program generates the structure subtypes ProgNoteDoc, ProgNoteSect, and so on, but it does not generate the structure subtypes MedNoteDoc, MedNoteSect, and others. This avoids semantic inconsistencies that may occur when we import later another subtype of MedNote that may not have the corresponding structure subtypes. On the other hand, the integrated system does allow user to explicitly specify the generalized structure subtypes; for example, as in Figure 4, the user may create the type MedNotePar, and make DisSumPar and ProgNotePar its subtypes.

3.3 The Use of Functions

3.3.1 Querying

The basic querying of text objects in the integrated system is conceptually performed by a set F of functions. Each function f_p in F is associated with a partial tree pattern p, which describes a certain logical text structure. The result of application of f_p to a text object is TRUE if it is a minimal text object that satisfies p (i.e., no subcomponent satisfies p), FALSE otherwise.

To specify a function in F, the user makes use of some high order OSQL functions. [2] Each TM operator has a corresponding high order OSQL function. For example, the OSQL function par corresponds to the TM operator W/PAR. For convenience, we call these high order functions TM functions. In the simple case, a TM function takes an arbitrary number of character strings as arguments and return a function in F. For example, in the query

select t for each TextObj t where par('headache')(t); – (Q1)

the par function takes the word "headache" as argument. It returns a function (in F) that maps a text object to TRUE if the text object is a paragraph with the word "headache" in it, FALSE otherwise. OIDs of text objects that satisfy the pattern are then returned by the select statement.

TM functions also take functions from F as arguments. For example, in the query[3]

select t for each MedNote t where par('headache', sent('artery', 'biopsy'))(t); – (Q2)

the second argument to par is a function, which is the result of sent('artery', 'biopsy'). The function returned by par is then a function (in F) that returns TRUE if its argument is a text object that is a paragraph containing the word "headache" and a sentence with the words "artery" and "biopsy." We can thus compose the TM functions to describe the tree pattern the retrieved text objects should satisfy.

Par and sent are second order OSQL functions. For text components with titles defined, we have third order OSQL functions:

select t for each ProgNoteSect t where sect('course')('headache', drug('tylenol'))(t);

[2]A high order function is one that returns a function as result.

[3]Note also the use of the type MedNote in the query to limit the scope of the results to only medical note text objects.

Here, sect is a function that takes a character string and returns a second order function. A retrieved section must have the word "course" in its title. Also note the use of the TM function drug that corresponds to the attribute operator DRUG.

The user does not need to define these TM functions explicitly. They are implicitly defined as the TM databases are imported into OpenODB. Consequently text types imported from different schemas have different TM functions defined. The importation automatically extracts necessary information from the DDF for translating TM functions to some external function for execution (see Section 4.2.2). We also have built-in functions that correspond to basic TM operators OR and W/n: the choose and near functions.

3.3.2 Navigation

Functions are also used to traverse the text structure hierarchies. We have four basic navigational functions: up, down, prev, and next, each being an external function. Each takes (the OID of) a text object, and based on the text structure hierarchy it belongs to, returns the OID of another text object: up returns the parent of a text object, down returns the first child, prev returns the previous sibling, and next returns the next sibling. The semantics of these functions are based on the actual document structure, and not on the structure hierarchy. For example, given the progress note structure hierarchy in Figure 4, the next of a paragraph is the next paragraph in the document, regardless of whether they are in the same section.

We can further define other traversal functions that provide convenience. For example, we may define children using down and next.

3.3.3 Information Fusion

The character string arguments to the TM functions can be the result of an ordinary OSQL function. This gives us a way to join the data stored the database with the information stored in the documents in the text retrieval system. To illustrate, we revisit the example (Q3) discussed in Section 1. We have the type Physician with attributes name and specialty in our database. Consider the following query:

```
select t for each ProgNotePar t, Physician p
where
    par(name(p), 'bronchitis')(t) and
    specialty(p) = 'respiratory';                          – (Q3)
```

When this query is processed, the names of physicians with specialty "respiratory" are used in queries submitted to TM. The results are progress note paragraphs that contain the name of a physician whose specialty is "respiratory" and the word "bronchitis."

Besides using structured data in text retrieval queries, we may do the opposite – extract data from documents and bring them into the structured database. The results of the TM querying and browsing functions are OIDs that reference text objects. Thus we may store these OIDs in the database and follow the references later. Another way to extract the data is to retrieve the actual text. We define the external function content that takes a text object OID and returns the corresponding piece of text. Its use is demonstrated in this example:

```
select content(t) for each ProgNoteSent t
where sent('artery', 'biopsy')(t);
```

The text strings returned may then be assimilated into the database directly or after further massaging.

3.3.4 Beyond Tree Inclusion

The processing of TM functions is integrated with the general query processing of OpenODB. The power of the two systems combined allows us to perform querying not possible with any one of them. Specifically, we are able to ask queries that cannot be expressed with the tree inclusion primitive. For example, consider again query (Q4) mentioned in Section 1:

```
select t for each TextObj t, TextObj s
where
    par('heart failure')(t) and par('headache')(s) and
    (next(s) = t or next(t) = s);                              – (Q4)
```

Here, we are asking for paragraphs with the phrase "heart failure," which are adjacent to paragraphs with the word "headache." We cannot express this query in TM or using the tree inclusion primitive alone.

4 Implementation of the Prototype

Figure 5 shows the various components in the prototype. The Text Objects Manipulation Module (TOMM) is a module that we have implemented (in C) to provide the desirable object abstraction of the text retrieval system. The text querying and browsing functions are implemented as external functions that access TOMM. External keys returned by TOMM are converted algorithmically into OpenODB OIDs. Below we discuss each aspect in detail.

4.1 Text Objects Manipulation Module

Although the query language of TM allows an arbitrarily fine granularity of content specification (depending on the mark-up of the documents), it does not support the notion of text objects in

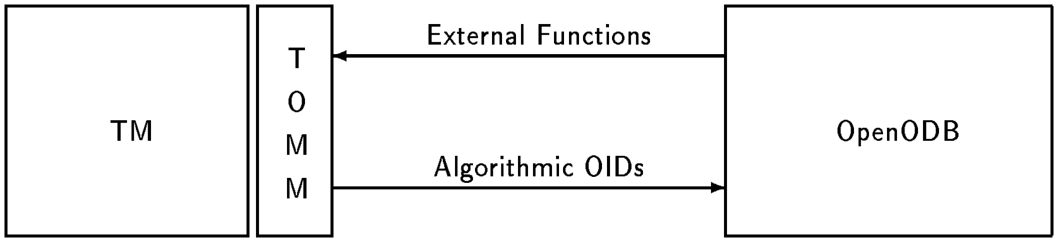

Figure 5: Overview of implementation

the retrieval, i.e., the unit of retrieval result is still a document – a set of entire documents are returned. Thus, to support a model of text objects in our integration, we built an encapsulating module, called the Text Objects Manipulation Module (TOMM), to objectize TM.

TOMM identifies the individual components of a document, or text objects. Each text object is represented by a tuple of eight keys, which divides into three parts. The first is the document identifier of the document that contains the component (it is in turn made up of three integers: the database number to which the document belong, the partition number within the database, and the document number within the partition). The second part corresponds to the starting and ending positions of the component within the document (each made up of two integers). Finally, the last part is a text-category dependent key that represents the text object structure type (e.g., whether it is a sentence, a paragraph, or a chapter). With these eight keys, we can uniquely identify the text objects. (Actually, we do not need the last key to identify the object, but it is useful in supporting type checking in OpenODB.)

Query results from TM, which are in the form of a set of TM document identifiers, are passed through TOMM. TOMM first locates the components that contain the hit words in the query. It then derives the starting and ending positions of the components, looks up the structural keys, and returns a set of tuples as results. Section 4.3 discusses how these keys are converted to and from OpenODB OIDs. Given a set of keys, TOMM also performs the opposite process of retrieving the actual text of the component. Function content can thus be implemented.

4.2 External Functions

4.2.1 Navigational Routines

The external functions up, down, prev, and next have four corresponding supporting routines defined in TOMM. Each of the four routines takes as arguments the external keys of a text object, and returns the external keys of another text object. To illustrate, consider the routine for supporting the next function. Given the external keys, the routine first locates the component. It then searches the first start and end tags that appear in the document after the component. The keys for this component are then derived.

4.2.2 Function GetTextObjs and Query Translation

In the integrated system, the user queries text data with high order OSQL functions such as par and sect. These TM functions have no real implementation: instead of defining each and every TM function individually, we implement a single external function GetTextObjs. It acts the entry point to query TM. A predicate expressed with high order TM functions is translated into a call to GetTextObjs. Below we first discuss the function GetTextObjs and then we explain the query translation process.

GetTextObjs takes a variable number of character strings as arguments. The first argument is special: it is the skeleton of a TM query. It has special substitution symbols "%s" embedded, in the style of the printf statement in C. During query processing, the rest of the arguments are substituted into the first, forming a complete TM query. The result of GetTextObjs is a set of text objects retrieved by the TM query. Consider the following query:

```
select t for each ProgNotePar t, Physician p
where
    t occurs in GetTextObjs('(W/PAR[%s bronchitis])', name(p)) and
    specialty(p) = 'respiratory';                                   – (Q5)
```

In processing this query, the name of each physician whose specialty is "respiratory" will be substituted into the main query, and TM will be invoked a number of times; e.g., one query to TM may be (W/PAR[john bronchitis]), if "john" is the name of just a physician. The results from the TM invocations are merged together and the OIDs of the matching paragraphs are then returned by the query. Occurs in is an OSQL function that tests the occurrence of an object in an aggregate.

When GetTextObjs is called, the registered C routine will invoke the appropriate TM library procedure with the query. A document set is returned by the TM procedure. The results are passed to TOMM, which returns the external keys. GetTextObjs then converts the external keys into an OID (detailed in next section). A set of OIDs is then returned.

With the GetTextObjs function, we already achieve the goal of accessing TM from OpenODB. However, such basic integration is not adequate for a number of reasons. The first is the ease of use. The syntax of GetTextObjs, with the substitution symbols, is not very convenient. The user has to know the TM syntax. Secondly, as the TM query is not interpreted by OpenODB, syntax errors in the TM query are only discovered at run-time by TM.

Thus, the syntax of high order OSQL functions is a better way for the user to express text retrieval queries in the integrated system. To process them, we modify the OpenODB query translator to translate them into the primitive GetTextObjs. Let us first illustrate this with an example. Consider query (Q1) again:

```
select t for each TextObj t where par('headache')(t);              – (Q1)
```

It is rewritten into the following for execution:

```
Function: convert
Input: a TM function call tree (func, arg[0], arg[1],...)
Output: ans, a list of arguments to GetTextObjs
Method:
    map func into the corresponding TM operator string and make it the first element
    of ans;
    for each argument arg[i]
        if arg[i] is a character string        /* case (i) */
            concatenate it to the first element of ans;
        else if arg[i] is a TM function call        /* case (ii) */
            call convert with the call tree recursively;
            let childans be the list returned;
            concatenate the first element of childans to the first element of ans;
            append the rest of childans to ans;
        else if arg[i] is a call with character string result type        /* case (iii) */
            concatenate "%s" to the first element of ans;
            append arg[i] to ans;
        else
            signal error;
```

Figure 6: Pseudo code for text object querying predicate translation

```
select t for each TextObj t where t occurs in GetTextObjs('(W/PAR[headache])');
```

Note the transformation is performed on the predicate par('headache')(t) in the where clause. In OpenODB query processing, after the parse tree of the query is built, there is a stage in which we process predicates in the where clause. At this stage, we check if the predicate is a TM predicate, i.e., a predicate that contains a TM query operation. If it is, instead of allowing the normal processing to continue, we intercept and replace the call tree with that of a call to Occurs in, whose first argument is the text object variable (e.g., t in the example), and the second argument is a call to GetTextObjs. The arguments to GetTextObjs are formed by transforming the TM function call tree. Finally, after the call tree is replaced, normal query processing resumes.

Figure 5 is the pseudo code for the function convert that does the transformation. (For simplicity of presentation, we omit the case for third order OSQL functions; e.g., sect('course')('headache')(t).) The input to convert is the call tree of a TM function (e.g., par). The output is a list of arguments to GetTextObjs. The first element of the output list is the main TM query; the rest are arguments to be substituted into the first.

The translation handles three different cases of arguments to TM functions. The first (case (i) in pseudo code) is for character string arguments. In this case, the string is just concatenated to the main query. In case (ii), convert is recursively called to handle nested TM functions; e.g., the function sent in query (Q2):

```
select t for each TextObj t where par('headache', sent('artery', 'biopsy'))(t);          – (Q2)
```

The query is transformed to the following:

```
select t for each TextObj t
where t occurs in GetTextObjs('(W/PAR[headache (W/SENT[artery biopsy])])');
```

Finally, in case (iii), the argument is an ordinary OpenODB function call with character string result type. It is made into an argument to GetTextObjs and a corresponding "%s" symbol is embedded in the main TM query string. This is how the call name(p) is handled here in query (Q3):

```
select t for each ProgNotePar t, Physician p
where
    par(name(p), 'bronchitis')(t) and
    specialty(p) = 'respiratory';                                                  – (Q3)
```

The query is rewritten to (Q5) for execution:

```
select t for each ProgNotePar t, Physician p
where
    t occurs in GetTextObjs('(W/PAR[%s (bronchitis)])', name(p)) and
    specialty(p) = 'respiratory';                                                  – (Q5)
```

The necessary information required to do the checking and translation is extracted automatically from the DDF during importation. The information is entered into some system tables and is used during translation.

4.3 Algorithmic OID Generation

In mapping keys from external sources to OpenODB OIDs, we have two options. The first is to create new database objects to correspond to the external objects. The database objects would have an attribute, say extkeys, to store the external keys. For example, in our integration, we could have created the type TextObj as follows (recall that the external keys from TOMM are a tuple of eight integers):

```
create type TextObj functions
(extkeys Tupletype(Integer, Integer, Integer, Integer, Integer, Integer, Integer, Integer));
```

The extkeys attribute stores the external keys of a text object. We also need the inverse function that returns an OID for a text object, given its external keys; an OID is created if the text object does not already exist in the database. This scheme is expensive in terms of both time and storage:

- the conversion involves expensive index lookups and potentially object creation, slowing down retrieval, and
- it is expensive to store these objects, as the number of text objects is potentially huge (e.g., each text object can be a sentence, or of even finer granularity).

A second option avoids these problems. In this scheme, we do not create any text objects in OpenODB. Instead, we use an algorithmic conversion technique: from the external keys, we directly compute an OID. The inverse of this computation is used to convert from an OID to the external keys. These OIDs do not represent any objects in the database; instead, they are just references to data stored in TM. No database system resource is used to store these OIDs. A basic assumption here is that the external keys for a text object are fixed, which is true if we only add documents to TextMachine (i.e., no deletion or update).

The algorithm to compute an OID is simple: we just fit the eight integers representing the external keys into the length of an OID, which is 64 bits. We distinguish an algorithmic OID from ordinary OIDs by setting some of the reserved bits.

We also need to modify the type checking system of OpenODB to recognize these OIDs. Basically, we must let the system know

- the type to which a particular algorithmic OID belongs, and
- the typing function of text types; i.e., given an algorithmic OID, determine if it is of a certain text type.

The necessary information to do type checking of text objects is entered into some system tables during DDF importation. The typing functions are also automatically generated. In compiling the queries, the type checking system consults these tables and uses the appropriate typing functions.

5 Related Work

5.1 Integration Aspects

A lot of past efforts have been made to blend text retrieval capability into database systems. Reference [15] investigates the efficiency of nested relational document database systems. In [13], Lynch and Stonebraker propose to extend relational database systems with user-defined indexing. Reference [12] discusses the integration of text retrieval capability into the ORION object-oriented database system. These are internal integration approaches: text indexing and retrieval capabilities are added to the database system internally.

A recent work [6] is related to our external integration approach. There, Croft et al. describe the integration between the inference net model retrieval system INQUERY and the database system Iris (Iris is the research prototype of OpenODB, the database system used in our integration). It is a loosely-coupled integration: the user interacts with a separate control module, which accesses both INQUERY and Iris through their application programming interface. Iris is used to store components of documents, and indexing and retrieval are done by the INQUERY system. The version of the Iris system used in their work had no external function capability, and thus the integration achieved is not as tight as in our case: there is no integrated query language or query processing. Performance, in terms of retrieval response time, is reported as a serious problem. In our work, to improve performance, we make use of external functions to achieve tightly coupled processing and investigate algorithmic OID conversion techniques. We also address issues relating to the modeling of text, especially under automatic importation of document schema. Their work assumes a particular type of text (Ph.D. thesis in LaTeX) and thus modeling issues are not addressed.

5.2 Structured-Text Retrieval

Fox [7] gives an overview of the problem of retrieving composite documents. A number of recent proposals have appeared to address the issue of querying hierarchical texts by partial patterns [4, 11,14]. These models support retrieval of structured-text components using a partial pattern, based on the containment criteria. The TextMachine system that we have integrated bears resemblance to these proposals.

6 Conclusion and Future Work

We describe the integration of a structured-text retrieval system (TextMachine) into an object-oriented database system (OpenODB). Our approach is a light-weight one, using the external function capability of the database system to encapsulate the text retrieval system as an external information source. Yet, we are able to provide a tight integration in the query language and processing; the user can access the text retrieval system using standard database query language. The object-orientedness of the database system allows us to faithfully model complex text categorization and structure hierarchies. The user can retrieve components of documents at fine granularity, browse documents based on their structure, and have querying power beyond that provided by the text retrieval system. The importation of new text types from the retrieval system is effortless; meta information is extracted automatically. The performance overhead of accessing an external source is alleviated by the technique of algorithmic object identifier conversion.

6.1 General External Integration Framework

The approach and techniques of external integration reported in this paper can be extended to other retrieval systems as well; as already mentioned, we have integrated OpenODB with the WAIS system, which has a rudimentary statistical ranking retrieval model. The structured-text retrieval system used in our integration uses exact-match tree inclusion. As structured-text retrieval systems based on statistical ranking become available to us, we believe we can incorporate them with minimal modifications in our framework.

It is interesting to further generalize the framework to integrate diverse information sources and answer questions such as: What is the desirable abstraction for the information source? What meta information are needed? What is the minimal set of functionalities that the information source has to provide?

6.2 Optimization in the Presence of External Functions

Another interesting issue is query optimization involving external functions. Recent work [5] has been done on this topic, but external functions for text retrieval present a specific and yet important enough scenario for further investigation. For example, in processing the external function calls to TM, when there are multiple calls to GetTextObjs (joined by OSQL boolean operators) in the same select query, we may choose to combine the GetTextObjs calls into one, provided they are bounding the same text variable. Complications arise when the calls involve arguments that are bound by other OSQL predicates. We plan to look into this issue in the future.

Acknowledgement

Thanks to Rafiul Ahad, Umesh Dayal, Dan Fishman, Hector Garcia-Molina, Catherine Hamon, Ralph Keiser, William Kent, Mary Loomis, Roger Miller, and Anthony Tomasic for reading drafts of this paper and providing helpful comments.

References

[1] Alliance Technologies, *TextMachine User's Guide and Reference Manuals.*

[2] J. Annevelink, R. Ahad, A. Carlson, D. Fishman, M. Heytens, and W. Kent, "Object SQL – A Language for the Design and Implementation of Object Database," to appear in *Database Challenges for the 90's*, W. Kim ed., ACM Press, 1994.

[3] W. Appelt and N. Tetteh-Lartey, "The Formal Specification of the ISO Open Document Architecture (ODA) Standard," *Computer Journal*, 36(3):269-79, 1993.

[4] F.J. Burkowski, "An Algebra for Hierarchically Organized Text-Dominated Databases," *Information Processing & Management*, 28(3):333-48, 1992.

[5] S. Chaudhuri and K. Shim, "Query Optimization in the Presence of Foreign Functions," *Proc. VLDB Conference*, Dublin, Ireland, 1993, pp. 529-42.

[6] W.B. Croft, L.A. Smith, and H.R. Turtle, "A Loosely-Coupled Integration of a Text Retrieval System and an Object-Oriented Database System," *Proc. ACM SIGIR Conference*, Copenhagen, Denmark, 1992, pp. 223-231.

[7] E.A. Fox, "Composite Document Extended Retrieval," *Proc. ACM SIGIR Conference*, Montreal, Canada, 1985, pp. 42-53.

[8] C.F. Goldfarb, *The SGML Handbook*, Oxford University Press, 1990.

[9] Hewlett-Packard, *OpenODB Reference Manual B3185A*, 1st ed., 1992.

[10] B. Kahle and A. Medlar, "An Information System for Corporate Users: Wide Area Information Servers," *Connexions – The Interoperability Report*, 5(11):2-9, 1991.

[11] P. Kilpelainen and H. Mannila, "Retrieval from Hierarchical Texts by Partial Patterns," *Proc. ACM SIGIR Conference*, Pittsburgh, 1993, pp. 214-222.

[12] W.L. Lee and D. Woelk, "Integration of Text Search with ORION," *IEEE Data Engineering Bulletin*, (13)1:58-64, 1990.

[13] C.A. Lynch and M. Stonebraker, "Extended User- Defined Indexing with Application to Textual Databases," *Proc. VLDB Conference*, Los Angeles, 1988, pp. 306-317.

[14] I.A. Macleod, "Storage and Retrieval of Structured Documents," *Information Processing & Management*, 26(2):197-208, 1990.

[15] J. Zobel, J.A. Thom, R. Sacks-Davis, "Efficiency of Nested Relational Document Database Systems," *Proc. VLDB Conference*, Barcelona, Spain, 1991, pp. 91-102.