Integrating Structural Search Capabilities Into Project Haystack

by

Svetlana Shnitser

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

Author ....
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by ....
David R. Karger
Associate Professor
Thesis Supervisor

Certified by ....
Lynn Andrea Stein
Associate Professor
~~Thesis~~ Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Integrating Structural Search Capabilities Into Project Haystack

by

Svetlana Shnitser

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

In this thesis, we have designed and implemented a system for performing structural searched in Haystack. The Haystack data model is semi-structured, and the challenge of this project was to develop a system that performs database-like queries on semistructured data using current relational database technologies. To achieve this goal, we have designed a database schema that would allow us to store our data model. We have specified the format in which the user can enter database queries and implemented procedures that translate user queries into SQL. We have designed a way to integrate structural search with text search in Haystack and have outlined ideas on how database queries can be used for machine learning.

Thesis Supervisor: David R. Karger
Title: Associate Professor

Thesis Supervisor: Lynn Andrea Stein
Title: Associate Professor

Acknowledgments

First and foremost, I would like to thank David Karger and Lynn Stein for making this thesis possible in the first place. I am grateful for their guidance and support; for taking the time to discuss and improve our designs; for their enthusiasm about the project and their abundance of ideas — I wish I had the time to explore them all!

I would also like to thank Wendy Chien with whom I have been working closely on Haystack for the past few months — it's been fun working together. Thanks to Ziv Bar-Joseph and Jaime Teevan for keeping us a good company at the office. Thanks to Ian Lai for making last summer very productive, and for introducing many improvements and adding useful features to Haystack.

Haystack is a team effort, and I would like to thank everyone who contributed to the project - Adam Holt, Adam Glassman, Damon Mosk-Aoyama, Ken McCracken, Will Koffel - it has been my pleasure being part of the group.

Finally, I would like to thank my boyfirend Sasha for his love and for making sure that our lives are not only about work.

The past year has been an enjoyable time and I am grateful to everybody who made it so.

Contents

List of Figures

Chapter 1

Introduction

As more and more aspects of life have moved online, the amount of digital information the user accumulates and processes has grown tremendously. From personal emails, to messages containing one's phone bills or airline reservations, to online news articles - the data is becoming increasingly difficult to manage and keep track of. Until recently, searching through personal information space was not a significant problem, as the amount of personal digital information was limited and the user could get by with the simple search tools such as Windows Find tool, or Unix grep. However, as more and more information and communication take place in a digital form, these simple techniques have become ineffective. The Haystack project is an attempt to create a personal information retrieval system that would help the user search through and organize his body of knowledge.

With the proliferation of digital information came a variety of search engines. Most of them, however, are designed to locate the information on a particular subject among all the documents available on the web. Unlike web search tools, Haystack is meant to work with a local corpus of information that the user accumulates. Haystack can handle a variety of documents - from emails in the user's mailbox, to the web pages he visited, to the papers he has written or read in almost any format.

Not only does Haystack know about the documents' contents, but it also records the documents' metadata, such as the author, the subject, or the date. This allows the system to answer very specific queries about the documents in one's collection. Finally, Haystack adapts to the individual user by keeping track of which documents the he finds relevant and using this knowledge to produce better search results.

1.1 Types of Queries

What kind of questions might the user ask about his own documents? For example, he may be planning a trip to France and want to find all the documents he has accumulated about Paris. "About Paris" is a condition that is best be handled by a text search engine that does a full-text fuzzy matching. In general, text search engines are good at answering "about" queries - they can deduce whether or not a document is "about" some something by examining how often the query terms occur within a document. On the other hand, the user may be interested in a document entitled "Machine Learning" that Wendy has sent to him 2 days ago. The constraints of the second query are different from the conditions of the first — they are exact specifications on what the title and the date of the document should be, rather than a vague term that the document should be about. These types of queries are typically handled well by a search engine that can keep a table of all the attributes of the documents that it works with. An example of such search engine is a relational database. On another occasion, the user may want to find the documents similar to the one he has just read. This kind of query requires associative search and is best handled by a system that keeps track of relationships between documents.

Thus, we can see that at different times, the user may have different information needs and constraints, and will require different types of systems to satisfy those needs. We call this search that involves different types of information systems a *hybrid search*. One goal of the Haystack project is to create a system that allows users to perform hybrid search. Haystack is not meant to be a new search engine; rather, it's designed to be an interface to various types of information retrieval systems. When a user enters a query, haystack should analyze it, dispatch appropriate requests to the underlying search systems that best match the user's needs, and afterwards intelligently combine the results returned by those systems. For example, if the user requests "all documents about California that he has read in the last week," Haystack should dispatch the task of finding all documents about California in the user's repository to a text search engine. The second part of the query asks Haystack to limit the result set to a certain time period and should be dispatched to a database. Afterwards, Haystack should return the results that match both conditions.

1.2 Goals of This Thesis

It was originaly proposed by Adar in [1], that Haystack combines 3 types of information systems:

- a text search engine
- a database-like search
- a system that performs associative search.

When the author started her work on the project, the text search engine and the associative search were incorporated into Haystack. The goal of this thesis was to design and implement a system that allows the user to perform database-like searches in Haystack, and to integrate it with existing text search capabilities to make Haystack a hybrid-search system.

A relational database usually maintains a table storing the data according to some predefined schema. This works well when the data is structured (i.e. all instances of it have the same set of attributes.) One of the challenges of adding database-like search capabilities to Haystack lay in the fact that our data model is semi-structured, which means that the set of attributes is not at all guaranteed to be the same across all objects and can dynamically change as the new objects are being archived. In our solution, we chose not to create our own database system tailored to semistructured data. Instead, we decided to use existing relational database technologies to "emulate" structured search on semi-structured information. We distributed the data in the database in a way that would allow the user to retrieve the objects based on a given set of attributes, without limiting the attributes or enforcing the structure that the objects may have.

The task of implementing structured search and integrating it into the current system consisted of several steps.

- We had to design a database schema according to which the data is be stored, and make sure that the information in the database is consistent with the information stored in the data model.
- We had to design the user interface, specify the format in which the user can formulate the queries, and devise a method of translating the queries from the input format into SQL
- Our last task was to incorporate database queries into the current query architecture in Haystack and integrate database search with the machine learning module that was being developed at the same time.

1.3 RelatedWork

1.3.1 Semi-Structured Search

The most important project to which this thesis is closely related is project Lore (Lightweight Object Repository) which is being developed at Stanford [5]. Lore's objective is to explore the issues associated with storing, querying, and managing semistructured data. It includes a database management system tailored specifically to semi-structured data, a language for formulating the queries, and a system called DataGuide that creates a structural summary of the underlying data and allows the user to browse the material stored in the repository.

It was originally proposed in [1] that we use Lore as our data repository and as a semi-structured search tool. However, we later changed this deicision. Being a research project, LORE gives no guarantee that all the functionality will be in place and will be supported in the future. However, we chose to include some of Lore's ideas in our design. In particular, we adopted Lore's model for the query language, as it seemed convenient to express a wide range of queries about theattributes and structure of the user's data.

1.3.2 Personalization.

As the amount of information on the internet has increased drastically, personalization of search and content has become a popular trend that gives users a "more tailored, efficient Web experience." Numerous web sites create user profiles by identifying users each time they visit a site, recording their preferences and then delivering ads and content targeted to their profile. On the personalized content side, perhaps the most popular example is myYahoo [15] — it allows the users to customize their Yahoo front page to display only the information that is interesting to them.

A slightly different approach to personalization is taken by the DirectHit web search engine [?]. Instead of monintoring individual user's information needs, DirectHit works by ranking search results according to their relevancy with previous searchers who share similar personal traits. To create "personalized" results, directHit solicits anonymous demographic information such as the age, gender, and geographic location of their customers. This information is then used in the ranking algorithm to further refine the search results. DirectHit's approach is similar to Haystack in that Haystack also uses relevance feedback techniques (on a per-user basis, though) to improve the rankings.

Among the personalized search tools, an example would be an application created by PurpleYogi [17]. PurpleYogi attempts to learn people's interests to bring them just the articles and other resources they want. To that task they added a new requirement: the system must work without keeping central records of customers' interests and activities. The PurpleYogi software, a free download, sits on the user's hard drive and automatically builds a profile of the user's browsing habits; this profile is stored only on the user's machine, and proactively searches the Internet for more sites of potential interest to the user. PurpleYogi is not a search tool, but its ideas are close to Haystack, since Haystack is also indended to monitor user's interests and infer which documents are relevant to those interests.

Another information retrieval tool similar in spirit to PurpleYogi is a Remembrance Agent [9]. It is a program that continuously displays a list of documents which might be relevant to the user's current context. Unlike most information retrieval systems, the RA runs continuously without user intervention. It suggests information sources which may be relevant to the user's current situation in the form of one-line summaries at the bottom of the screen. The full text of a suggestion can be brought up with a single keystroke.

1.4 Thesis Outline

Chapter 2 of this thesis provides the background information about Haystack, different search techniques, and presents a more thorough discussion of Lore. In Chapter 3, we discuss the format in which the user can specify the queries. In Chapter 4, we present our database schema, and in Chapter 5, descrive how to translate the queries from the user format to SQL. Chapter 6 talks about persistent storage and what we need to do to keep the data in the relational database consistent with data in the storage, and in Chapter 7, we present our ideas on how we can integrate structural search with text search and machine learning techniques. Chapter 8 desribes the architecture of the query system in Haystack. Finally, in Chapter 9 we outline future work. Chapter 10 concludes the thesis.

Chapter 2

Background

In this chapter, we will present the background information relevant to the topics discussed in this thesis. First, we will familiarize the reader with the Haystack system by describing the data model that we use to store the data and the general architecture of the system. Next, we will outline the basic principles of unstructured and structured search. Finally, we will talk about semistructured search and discuss the Lore project in greater detail.

2.1 Overview of Haystack

2.1.1 Haystack Data Model

All information in Haystack is represented as a labeled directed graph (also referred to as the data model, or data graph.) The nodes in this graph are called *straws*, and every straw has a unique ID. A *needle* is a subtype of straw and is used to represent a node that contains a piece of data, such as a String or a Date. The links between the nodes, called *ties*, are first-class objects and are themselves a subclass of straw. A tie represent a directed relation between the two straws that it connects. For example, if a straw A represents a document and straw B represents the type of that document, then the relationship can be expressed through a 'DocType' tie from A to B. The reason that relations are represented as independent nodes in our data model as opposed to being simply pointers is that we might want to point to or annotate a relation. For example, if the user has created a 'RelevantTo' link from one document to another, he may want to also explain the reason he created that link (i.e. annotate the tie.) In order to do that, he may attach to this tie a String object containing the explanation.

A *bale* is another subclass of straw and is a centerpiece that represents an n-ary relation between straws. A bale can be used to represent, for example, a document, a person, or a query, e.g. if we have a postcript document called Design.ps, created on 12/02/99, it will be represented in the data model as follows:

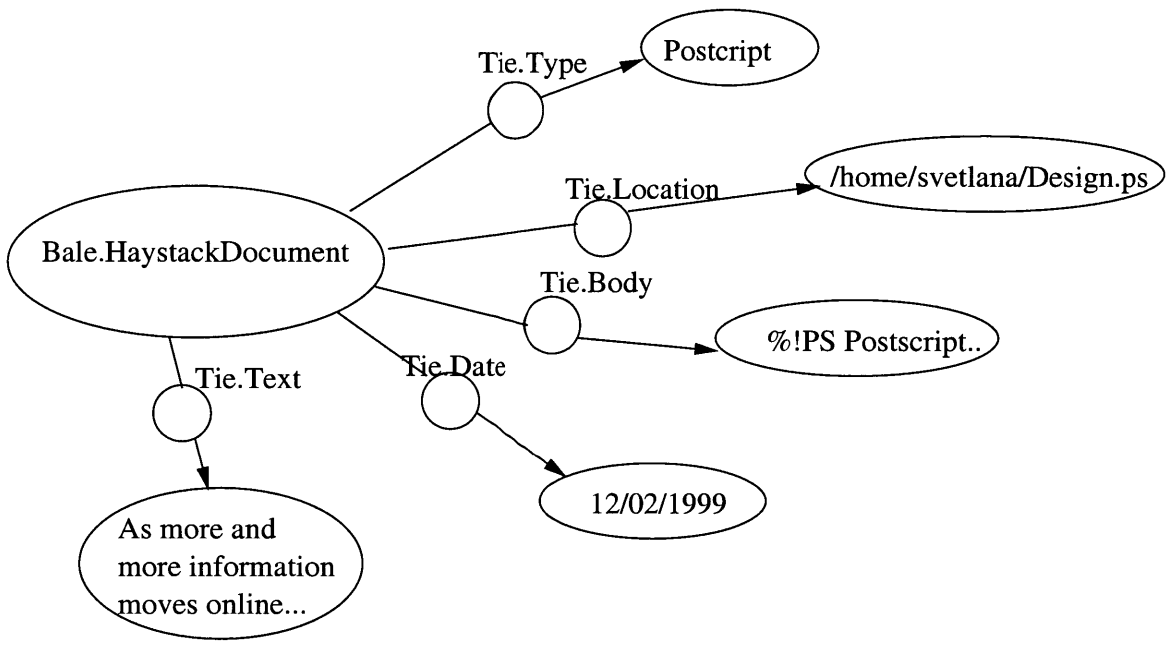


Figure 2-1: Data Graph Representing a Document Bale

Note that the label of the tie only describes the relationship between two nodes, and does not imply anything about the type of the object that the tie points to. For example, we can have a Tie.References pointing to another document bale, or it may point to a needle containing the title of the referenced document. While this approach adds flexibility to our data model (Tie.Date may point to a Date needle or to a needle saying "MyBirthday"), it also precludes us from making potentially useful assumptions about the contents of the needles.

It is important to point out that Haystack data graph contains not only the documents that get archived, but virtually all information we use in Haystack. For example, when a user issues a query, we create a query bale that contains the query string and has a "MatchesQuery" tie poiting to all the Straws that were returned as results to the query. The reason we keep queries as part of our data model is that we want to be able to point the user to his past queries, especially if they seem relevant to his current query. Similarly, all Haystack services (which we'll talk about below) are also represented as bales in our data model, so that every straw that a service creates has a Tie.Creator pointing back at the service. Another reason to keep "auxiliary" information in the data model is that a lot of services are data-driven, i.e. they are triggered when particular nodes are added to the data graph by another service. Thus, it may be convenient to keep this auxiliary information in the data model if we want more than one service to be aware of it.

2.1.2 Haystack Architecture

In general, the Haystack architecture follows a client-server model, where the server is reponsible for managing the data, and the client, which can run either locally or on a remote machine, is the interface through which the user can browse or query his Haystack, or archive new documents into it. On the server side, Haystack consists of a family of individual services, which are "functional units" of Haystack and have a well-defined set of duties.

Kernel services are at the heart of Haystack. They include `HaystackRootServer` which is responsible for bootstrapping the system, the `NameService` which keeps track of all existing services, the `Dispatcher` that knows when to trigger services, and the `PersistentObjectService` that is responsible for creating and storing the data graph.(See Appendix A for an in-depth discussion of kernel services.)

Data Manipulation Services are responsible for processing and archiving the documents. They are data-driven, i.e. triggered by an addition of a straw to the data graph. They include the `TypeGuesser`, the `Archiver`, and various extractors, unwrappers, and textifier services that extract individual files from collections (such as directories or tar archives), translate them into the correct format (if they are UUEncoded, for example) and extract text from them for the purposes of indexing.

Haystack also includes Communication services that are responsible for passing the data between the client and the server, ObserverServices that track the user's web browser and mail client to actively archive the documents that he works with, and various other services not mentioned in this thesis. For a full discussion of Haystack architecture and services, consult [1, 2]. For additional background information on Haystack, see [14].

2.1.3 Example of how Haystack works

Rather than spend time describing various services, let's illustrate how Haystack works with an example. Suppose the user archives a postscript file at located at */home/svetlana/Public/thesis.ps*

First, the archiver gets to work and creates a bale for this document (using ObjectCreatorService) and attaches to it a Location tie pointing to the needle containing *"file://home/svetlana/Public/thesis.ps"* Next, the TypeGuesser service is triggered by the Dispatcher. The TypeGuesser determines that this document is of type postscript and attaches DocType tie to the needle representing the postscript type.

Once the type has been determined, the Dispatcher calls HsPostscript, since this service is interested in document bales that have a Tie.DocType leading to a Needle containing "postcript." HsPostcript extracts the text from the file and creates Tie.Text pointing to the needle that contains the text. Finally, HsIndex is called — it is triggered by the creation of the Tie.Text. HsIndex, as the name suggests, indexes the text of the file to be used for text searches.

Of course, the steps above are only a rough outline — a lot more things happen in Haystack as the document gets archived — but this example should give a flavor of how Haystack operates.

2.2 Search Techniques.

2.2.1 Text Search.

Text search is a type of search where the user specifies one or more keywords that he thinks should be contained in the document, or should reflect what the document is about. The general idea behind unstructured search is the use of an inverted index file that maps words to the documents in which those words appear. When a user issues a query for "word1 & word2", the system looks up the set of documents that contain both of those words. This simplistic type of IR system is called a Boolean system. More sophisticated IR systems combine boolean search with ranking. Those systems project every document into n-dimensional vector space, where n approximately corresponds to the number of words in the language, and record how often each word occurs in a given document. When a query is issued, a vector corresponding to the query terms is created and thrown into the vector space. The IR system then uses some metric (usually the dot product) to find the documents whose vectors are closest to the query vector (they do not have to contain all the words in the query, however). The assumption behind this approach is that the more words the document shares with the query, the better match it is.

2.2.2 Structured Search.

The major contrast between structured and unstructured search is the items we are trying to categorize and index. While the unstructured IR system concentrates on

SSN	FirstName	LastName	Position	City
512687458	Joe	Smith	Manager	Sunnyvale
758420012	Mary	Scott	Engineer	Mountain View
102254896	Sam	Jones	Engineer	Cupertino
876512563	Sarah	Ackerman	Intern	Sunnyvale

Figure 2-2: EmployeeTable

the act of "indexing" the data only with respect to the language features, structured search systems record specific attributes of each object and require that all instances of the data conform to the same pattern. Traditionally, structured search tools fall into the domain of relational database technology, and since databases understand data types, it is possible to formulate powerful queries about information other than text.

Relational Databases

A relational database is a collection of two-dimensional tables. Each table contains a set of records all of which conform to a predefined schema that dictates the data type of each column in the table. A sample table may look like Figure 2-2. Structured Query Language (SQL) allows the users to define, manipulate, and access the data in a relational database. For example, if we wanted to list the first and last names of all engineers, we would formulate it in SQL as follows:

```
SELECT FirstName, LastName FROM EmployeeAddressTable
          WHERE Position = 'Engineer'
```

This should yield:

FirstName	LastName
Joe	Smith
Mary	Scott

The 'select' clause specifies the attributes (the columns of the table) that the query should select. The 'from' clause lists all the tables that are used in the query. The 'where' clause allows us to specify the conditions that the records should satisfy.

Of course, we are not limited to using equality in our where clause, and could use any of the relational operators such as LIKE (regexp), BETWEEN (valuel, value2) and so on. The where clause can also consist of a boolean combination of conditions.

Keys and Joins.

A primary key is a column or set of columns that uniquely identifies the rest of the data in any given row. For example, in the EmployeeTable, the SSN column uniquely identifies each row. A foreign key is a column in a table where that column is a primary key of another table, which means that any data in a foreign key column must have corresponding data in the other table where that column is the primary key. For example, if we created a table called Salaries listing positions and corresponding salaries, then the Position column would be a foreign key of the EmployeeTable.

Salaries:

Position	**Salary**
Manager	$ 80,000
Engineer	$ 70,000
Intern	$ 40,000

Good database design suggests that each table lists data only about a single type of entity, and detailed information can be obtained by using additional tables. Foreign keys let us relate the data in 2 tables. For example, we could create a listing of employees and their salaries with the following statement:

```
SELECT FirstName, LastName, Salary FROM EmployeesTable, Salaries
        WHERE EmployeesTable.Position = Salaries.Position
```

This would yield:

FirstName	**LastName**	**Salary**
Joe	Smith	$80,000
Mary	Scott	$70,000
Sam	Jones	$70,000
Sarah	Ackerman	$40,000

This operation is called a join. It "joins" two tables by listing all possible pairs of records, and then selects the rows that match given condition. In this query we select the records where Position is the same in both tables. Note that both tables involed in the relation have to be listed in the From clause.

Views and Aliases

Two more features of SQL will come handy in reading this thesis — they are views an aliases. Views allow us to assign the results of a query to a new, "personal" table that can be used in other queries, and can be used to restrict database access or to simplify a complex query. For example, we could say:

CREATE VIEW SunnyvaleResidents **AS**
        SELECT FROM EmployeeTable **WHERE** City = 'Sunnyvale'

Then we can use SunnyvaleResidents as we would any other table:

SELECT LastName **FROM** SunnyvaleResidents
        WHERE Position = 'Manager'

Aliases are just a convenient way of renaming the tables. For example, if we do not want to type EmployeeTable and Salaries every time, we could say

SELECT FirstName, LastName, Salary **FROM** EmployeeTable ET, Salaries S
        WHERE S.Position = ET.Position.

What we have outlined above is by no means an exhaustive list of relational database and SQL features, but it should be sufficient to allow the user who is not familiar with SQL to be able to read through the discussions of databases and understand SQL statements presented in this thesis. For an SQL tutorial, see [8].

2.3 Semistructured Search and Lore

Traditional databases work well when the data strictly adheres to explicitly specified schema. However, in many cases applications have to deal with the data that exhibits some structure but is irregular and thus does not conform to a rigid pattern. Moreover, it may be difficult to predict the schema in advance since the structure of the data may be evolving or the new data not conforming to the old schema may be added. We can call this type of data *semi-structured.*

As we stated before, Lore project at Stanford [5] is a DBMS devoted specifically to semi-structured data. Since a number of things in the Haystack data model and in our design of the semi-structured search capabilities for Haystack bear similarity to the Lore project at Stanford, we will describe Lore here in more detail.

Lore Data Model

Just like the Haystack data, the data managed by Lore is not confined to a schema, and it may be irregular or incomplete. Similar to Haystack, Lore's data model is a labeled directed graph called OEM (the Object Exchange Model.) Figure 2.3 contains a sample Lore data model showing the information about Haystack Group. The vertices in the graph are objects. Each object has a unique identifier (like HaystackID) such as &5. Atomic objects have no outgoing edges and contain a value from one of the basic atomic types such as integer, real, string, gif, audio, java, etc. (very much like Needles in Haystack). All other objects may have outgoing edges and are called complex objects. Object &3 is complex and its subobjects are &7 and &8. The names on the links between objects (analogous to Tie labels) indicate the relationship between the parent object and the child. Names may also serve as aliases for objects and as entry points into the database.

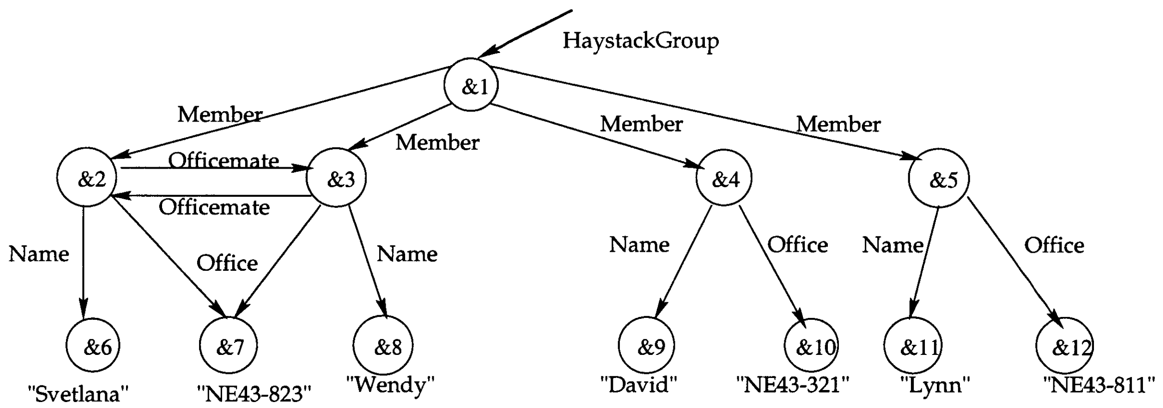


Figure 2-3: Sample OEM graph for Haystack group

The Lorel Query Language

In order to formulate the queries on the Lore data model, the authors have defined a query language called Lorel. Lorel is quite powerful and allows the users to express their queries as a form of "declarative navigation," that is, by specifying the query in terms of the paths through their data model and the values of the atomic objects.

In Lorel, a path expression is a name followed by a sequence of labels. For example, Person.Friend.Name is a sample path expression and consists of the set of objects that can be reached starting with the Person object, following an edge labeled 'Friend' and then following the edge labeled 'Name'. Range variables are similar to SQL aliases and can be assigned to path expressions, e.g. the user can say "Person.Friend.Name X" that would specifie that X ranges over all names of friends of people. Path expressions can also be used directly in an SQL style, as in the following example that refers to the sample data model:

SELECT DbGroup.Member.Office **WHERE** DbGroup.Member.Age > 30

Not only does Lore let the user to express the query in terms of the specific paths through the data graph, but it also allows for general paths expressions, i.e. the user can simply specify the patter for the path or for the atomic value, and the system will match it with all the paths that fit the pattern. A query involving patterns may look like

SELECT DbGroup.Member.Name
WHERE DBGroup.Member.Office(.Room% | .Cubicle)? like "%252"

Here the expression Room% is a label pattern that matches all labels starting with the string Room. | indicates disjunction between two labels, and the symbol ? indicates that the label pattern is optional. Basically , the syntax is based on regular expressions with syntactic wildcards such as "#" which matches any path of length 0 or more.

As we will see in the next chapter, we can adopt Lorel's notion of path expressions to our own format for specifying structural queries.

Chapter 3

User Input

In this chapter, we will discuss the user interface aspect of the structural query system. We will define a language similar to Lorel for formulating structural queries in Haystack and will present the web user interface. We will also introduce the notion of dataguides and describe how they can be used in our system.

3.1 Formulating a query

What kind of queries would the user like to pass to the database? As we mentioned earlier, the database contains the attributes of the documents (such as date, author, title), so a simple example of the database query would be "I would like to see all emails that I have received from John Smith in the last week." If Haystack had a natural language-recognition capability, the user would indeed be able to express the queries in English, and the sentence above could be parsed and translated into the appropriate query highlighting structured properties of the document sought. However, since language recognition is still a far-fetched goal, we needed to come up with a more formal way to express the queries.

The easiest approach would have been to input the queries in SQL itself - after all, SQL was designed specifically for querying databases. While it would demand no translation at all, this design would require the user to be aware of the way we chose to represent the data in the tables. Our goal was to create a user iterface that would intuitive, and we decided that it would be reasonable to assume that the user is familiar with the structure of the data graph. Hence, we chose to use Lorel's notion of path expression that we described in the previous chapter as a building block for our query format.

Let us look at the English version of the query again. In essence, it consists of three constraints that documents returned to the user should satisfy. The documents should :

- be of type email
- have a date of today−7days (assume 01-01-2000) or later
- be received from John Smith

Each constraint consists of an attribute (e.g. type, date) and a condition that the attribute should satisfy (=email, $\geq$ 01-01-2000, = 'John Smith')

If we go back to the Haystack data model, we can see that the attribute in essence corresponds to the tie leaving the document node, and hence the name of the attribute should have a direct correspondence to the tie label. So what the user is looking for is actually the following graph structure:

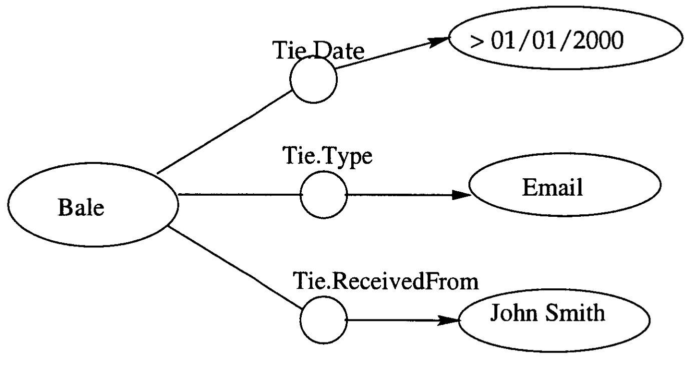


Figure 3-1: The Data Graph Corresponding to a Query

Using Lorel's query format, these constraints can be specified as:

SELECT Doc **WHERE**

Doc.Type = 'email' **AND**

Doc.Date $\geq$ 01-01-2000 **AND**

Doc.ReceivedFrom = 'John Smith'

Let's consider a slightly more complicated example. Suppose the user is looking for a document that references another document with the subject "Subject1." In terms of the data model, this would translate into a graph like this:

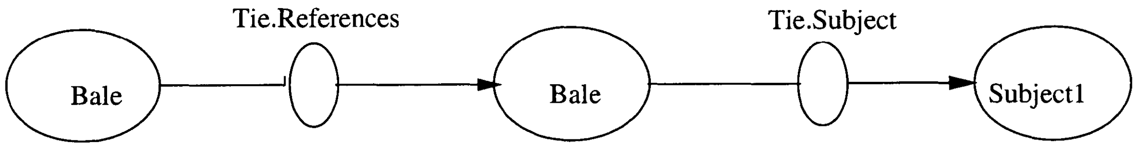


Figure 3-2: A Sample Data Graph for a Query Expressed in Terms of the Path through the Data Model

In this case, the attribute is more than just the label of the edge leaving the node - it is the path along several edges leading to the node that contains the data satisfying the condition. We can express this constraint as Doc.References.Subject = "Subject1"

So far we have assumed that the user is interested in having only document bales as results of the query and focused on expressing the constraints for these documents. We can actually relax this assumption — our notation allows the user to select not only the documents themselves, but also the attributes of those documents. For example, if the user wants to see all authors of the documents that have the subject "Haystack," he can express the query as:

SELECT Doc.Author.Name **WHERE** Doc.Subject = 'Haystack'

Similarly, a user can select several attributes at a time — selecting Doc.Author.Name AND Doc.Date where Doc.Subject = 'Haystack' should return return both attributes of the document bale.

We can even come up with syntactic sugar for expressing "inverse" attributes (i.e. specifying the ties that point to a node rather than leave it.) For example, if we want to find all the document that John Smith cites, we can say:

SELECT Doc **WHERE** Doc.(Cites).Author.Name = 'John Smith'

which should translate into

SELECT Doc.Cites **WHERE** Doc.Author.Name = 'John Smith'

One of the differences between Lore and Haystack data models is the fact that unlike Lore's edges, ties are themselves a subclass of straw and hence are nodes in the data graph. Lorel's path expressions do not allow us to formulate queries that involve ties connected to other ties. For example, if we have a document bale with the Author tie pointing to 'Svetlana' and an annotation on that tie pointing to a needle that says 'David thinks so', we cannot formulate a query that would express this graph structure.

For the first incarnation of our system, we decided to limit the expressiveness of our query language to a subset of features that Lorel provides. In particular, our query language currently does not include the notions of range variables and expressions containing wildcards.

3.2 Combining conditions

It is reasonable to assume that we would like to allow the user to type in a query that includes more than one condition. For example, he may want to search for recent email from customer service at his online bank about his new account. In this case he may formulate the query as follows:

SELECT Doc **WHERE**

Doc.From = 'customerService@mybank.com' **AND**

Doc.Date $\geq$ '01-01-2000' **AND**

Doc.Subject = 'New Account'

The most straightforward and strict way to answer this query would be to return the intersection of straws satisfying each constraint. However, we chose to approach this query from a different perspective. Instead of assigning each document a 0 or a 1 based on whether or not it matches the whole query, we assign it a score based on what proportion of individual constraints it satisfies . Then, we rank the documents according to their score and return the highest-ranking ones to the user.

This approach is more flexible that the straightforward intersection - if the user has made a slight error in one of the constraints (misspelled the name, for example), exact match may not return any results at all, whereas our system would still return the documents satisfying all other conditions, which would likely include the particular document that the user was looking for. Also, as we will see later, the varying grades of answers that this method produces may make it easier to combine database scores and text search scores to produce a new ranking.

3.2.1 Possible Extensions to the User Interface

Currently, our system assigns equal importance to all constraints. In the example above, suppose the user is not sure whether customerService was in the email address or in the subject line, but he knows that the date on the document should be within the last week. A simple addition to our user interface would be to let the user specify the relative importance of each constraint. Then, our ranking would take into account not only whether or not the document matches each constraint, but also the weights for each constraint the the user has entered. We assumme that an average user may not want to bother adjusting the weights, so we may leave this feature optional or reserve it for "advanced" users.

Another option to add to our user interface would be the "+" sign, which in many search system is used to signify that a particular keyword or condition *must* be satisfied.

Of course, it also may be the case that the user needs to specify precisely how the conditions should be combined, that is, he may want to formulate a boolean expression of individual conditions. For example, he may want to find the documents that either match some conditions A AND B, or match C AND D but not both (a XOR, in other words.) If he merely specifies all four conditions, then our system would rank highest the documents that satisfy all of them, if such exist, which does not match the user's needs. In the initial version of the system, we required that database query consisted of "simple" conditions and did not allow boolean expressions since they required additional parsing of the query. In the future, however, we plan to add this functionality.

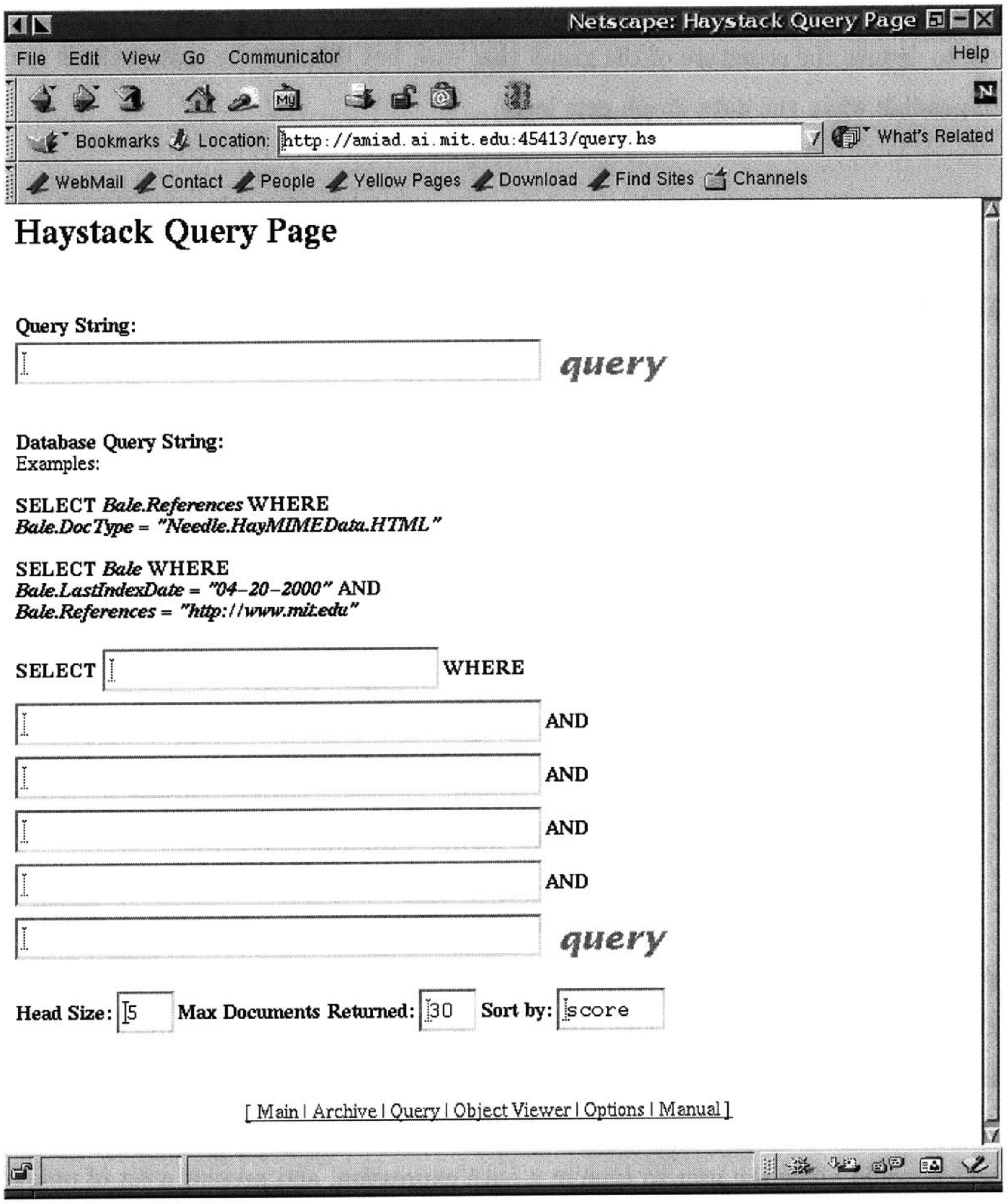
3.3 Web User Interface

Figure 3-3 contains a screenshot of our web user interface.

As seen from the figure, our user interface includes a text search box, where the user can type in his text query, and a set of boxes reserved for database search. Currently, we allow up to 5 individual conditions in the database query — this was a somewhat arbitrary decision which will not be hard to change if need arises.

3.4 DataGuides

The Lorel query format that we chose to adopt assumes that in order to form meaningful structured queries in Haystack, the user has to be familiar with the structure of the data. In traditional relational database system, the user just has to be aware of the schema to which all the data adheres. In Haystack, however, this is a more complicated task. Even though there are some regularities in the data graph, we


Netscape: Haystack Query Page
File Edit View Go Communicator Help
Bookmarks Location: http://amiad.ai.mit.edu:45413/query.hs What's Related
WebMail Contact People Yellow Pages Download Find Sites Channels

Haystack Query Page

Query String:

query

Database Query String:
Examples:

SELECT *Bale.References* WHERE
Bale.DocType = "Needle.HayMIMEData.HTML"

SELECT *Bale* WHERE
Bale.LastIndexDate = "04-20-2000" AND
Bale.References = "http://www.mit.edu"

SELECT WHERE
AND
AND
AND
AND
query

Head Size: 5 **Max Documents Returned:** 30 **Sort by:** score

[Main | Archive | Query | Object Viewer | Options | Manual]

Figure 3-3: The Web User Interface

impose no restrictions on the graph's structure, and in fact allow the user to add arbitrary links between the nodes, which means that we have no way of predicting how the graph will look. Of course, the user can simply browse his Haystack and try to deduce the structure of the graph that way, but this may be quite tedious and impossible when the data graph gets large.

Thus, it would be desirable to provide some "guidance" to the user that would help him formulate appropriate queries. Our solution is based on the notion of DataGuides [4] again, developed in the context of Lore. DataGuides are dynamically generated and maintained structural summaries of semistructured databases. In short, a DataGuide for a given graph contains all unique paths that are present in the graph. The figure below presents an example of a DataGuide for Figire 2.3:

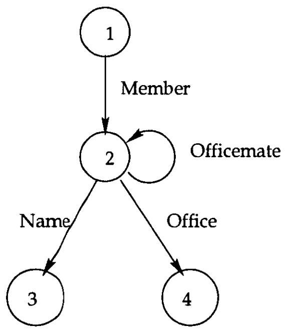
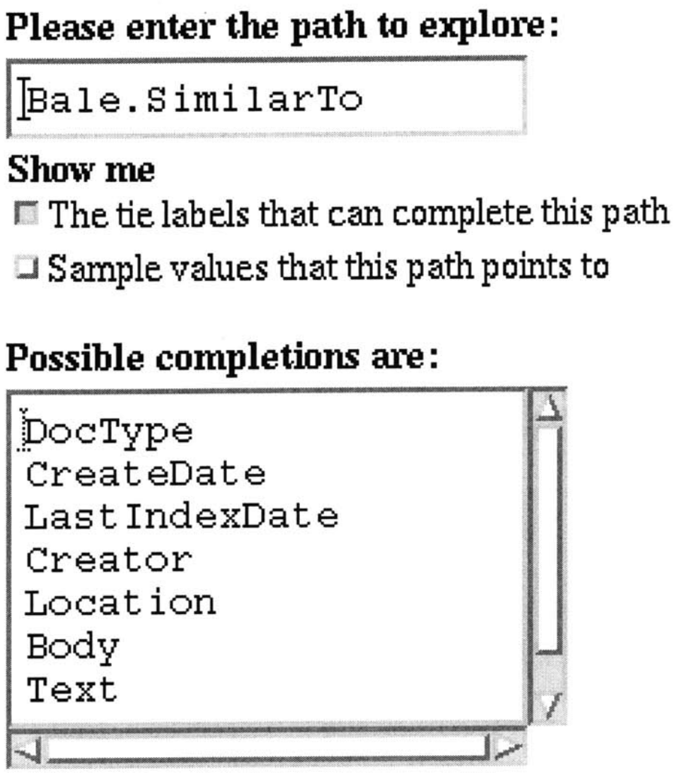


Figure 3-4: The DataGuide for the HaystackGroup OEM

In principle, we would like to give the user an option of browsing the complete DataGuide of his Haystack. However, building and incrementally updating DataGuides is a significant and non-trivial task (see [4] for a full discussion), so instead of implementing a full-featured DataGuide, we chose to provide "guidance on demand. " As an aid for query formulation, we decided to create an exploratory tool which would allow the user to type in a path expression, and present a set of unique tie labels that can continue the path, or a sample set of values that are contained in the needles that the path points at.

At the time of writing of this thesis, we have implemented the back-end functions


Please enter the path to explore:

```
Bale.SimilarTo
```

Show me

- ▨ The tie labels that can complete this path
- ☐ Sample values that this path points to

Possible completions are:

```
DocType
CreateDate
LastIndexDate
Creator
Location
Body
Text
```

Figure 3-5: A Sample User Interface for the Dataguide Tool

that would allow us to display the sample values or the labels that continue the path, but we have not yet incorporated DataGuide front end into Haystack. It would be desirable to have the DataGuide interface as part of Haystack query page (perhaps simply in a different frame). If the user selected one of the items in the list returned by the DataGuide, we could insert that value into the query box at the current cursor position.

Chapter 4

The Data Model for Semi-structured Search

In this chapter, we will address one of the key design issues of this thesis — how semistructured data can be embedded in a fully structured table for access by a RDBMS. We will present various alternatives and will discuss advantages and disadvantages of our design decision.

4.1 Designing the database schema for the data model

As we mentioned before, traditional database-management systems rely on the assumption that the data is strictly table-oriented and adheres to a schema that has been defined in advance. But according to our description in Chapter 2, the Haystack data model does not impose any tight restrictions or schema on the structure of the graph. It lets the user create any arbitrary links between the documents in his Haystack, and even though it is likely that most document bales will have certain attributes such as Date or Creator, this is not an intrinsic property of the data model. Moreover, different document bales may have completely different attributes - an HTML file will have "links to" attribute, while an email message will have "from", "to'", and “subject” attributes. Thus, even though our model allows us to store arbitrary objects and relations between them, it is clearly not tailored to a database.

Let’s examine our possibilities. It would be very convenient if we could structure the data in such a way that every straw can occupy a single row in our table, as in Figure 4-1. Then, we could just list all possible attributes (tie labels) as columns in that table. For each straw, we would fill in the attributes (list the contents of the needle pointed to by the tie with a given label), fill in the HaystackID if the tie points to another bale, or `null` if a straw does not have the attribute.

STRAW_ID	**DATE**	**TYPE**	**TITLE**	**FROM**	**TO**	**LOCATION**	**LINKS_TO**
5467	12/14/99	EMAIL	null	las@ai.mi..	svetlana@...	/home/ai/svetlana..	null
7856	01/23/99	HTML	MyHomepage	null	null	http://web.mit.e...	http://web.
9745	08/29/99	TEXT	Proposal	null	null	/home/proposal.txt	null
8564	03/15/00	POSTSCRIPT	Haystack	null	null	/home/haystack.ps	null
5367	04/30/00	EMAIL	null	karger@the..	haystack@the..	Mail/inbox/234	9745
...							

Figure 4-1: A Flawed Database Schema for the Haystack Data Model

Unfortunately, there are many problems with this approach. Even though the translation of “single-layer” queries (e.g. “SELECT Doc WHERE Doc.Title = “My-Homepage”) would be very straightforward, we may not be able to translate multi-layer queries since we are only storing the contents of the needles and not their IDs (i.e. this design assumes that all needles are leafs in our data graph, which is not true.) Furthermore, this schema implies that all needles pointed to by a given tie label have to contain the same type of data, since a database column has to be typed. This is exactly the assumption that we argued against in Chapter 2.

To resolve these issues, for every entry in the table, we could store the HaystackID of the needle pointed to by each tie rather than the needle’s contents (and store needles and their contents in a separate table.) However, this modification still leaves our design far from perfect. For example, the schema is defined in terms of the existing tie labels, so that if the user adds a tie we have not seen before, we would have to dynamically alter the table. Additionally, this design does not account for having several attributes of the same type — for example,a directory having multiple 'Contains' ties to each of the files in it. Finally, since many attributes are not shared across different types of documents, this table is likely to have many `null` values and thus would simply waste the space. A refinement on this approach could be to have different tables for each type of documents (one for emails, another for web pages etc.). While it would be more efficient in terms of space, it would still retain the other undesirable properties we outlined above.

Even though in the discussion of the Haystack data model we concluded that we are not imposing any constraints on our data graph, there exists a property it is guaranteed to have. Namely, that every tie is unique and has a single straw that it originates from and a single straw that it points to. In fact, we can use this property to create an acceptable schema for our database. Notice that the edges in the data graph are labeled (e.g. in Fig. 2-1, the link from the root node to the Author node is labeled Tie.Author), and tie labels are the key feature that will allow us to infer information about the structure of our data. In order to perform structural queries, we are going to represent the data graph in 2 tables: the first one (we will call it Ties) will store the relations between straws based on tie labels, and the second one (called Needles) will actually store the data contained in the needles themselves.

As seen from the Figure 4.1, the Ties table is simply a record of all the links in our data graph. The links are listed by their ID (the TieID column, which conveniently serves as a primary key since all ties are unique); the FromID and ToID contain HaystackIDs of endpoints of the tie, and the Label column, as the name suggests, contains the label of each link. The Needles table is a listing of the needles along with the data that they contain. We will discuss it in further detail later in this chapter.

Ties table:

TieID	**FromID**	**ToID**	**Label**
1435	1456	1457	Tie.DocType
1436	1456	1458	Tie.Location
1437	1456	1459	Tie.Date
2578	2729	2730	Tie.Author
2586	2729	2735	Tie.Title
2589	2729	2759	Tie.Score
...	...	...	...

Needles table:

ID	**Data**
1457	HTML
1458	http://haystack.lcs.mit.edu/...
1459	03-07-2000
2730	Svetlana Shnitser
2735	Integrating Structural...
2759	0.98
...	...

Figure 4-2: Database Schema for the Haystack Data Model

This arrangement of data seems particularly attractive because it places no constraints on the structure of the data graph - it merely records the graph in terms of the labeled edges and the contents of the nodes. A user can easily add a tie with a new label - and it would simply mean adding a new string in the Labels column. Likewise, since it treats every relation separately, there are no assumptions about the attribiutes that a bale may have.

As a variation on this approach, we could also create separate tables for each tie label that exists in Haystack. This way, we would not need to store the actual tie labels (which would be more space-efficient), since each table would contain only one kind of tie. It would also save us time spent selecting the rows with the appropriate tie label. The drawback of this approach is that translating the queries into SQL and managing the database would become more complex. We would have to keep track of as many tables as there are tie labels, and when translating the query, would need to make sure that every label that the user mentioned in his query is indeed valid and there exists a corresponding table for it.

4.2 The Needles Table

As we discussed in the previous section, the Needles table is intended to store the searchable contents of the needles in our data graph. Currently, there are 9 different needle types defined in Haystack:

- Needle.HayString is a wrapper around a Java String object.
- Needle.HayFloat wraps around Java Floats
- Needle.HayDate is a needle containing a Date object.
- Needle.HayURL contains Java URLs
- Needle.HayMIMEData wraps around an object used to store the mime type and encoding of a document
- Needle.HayFile represents a needle that contains a file
- Needle.HayByteArray is a needle containing arbitrary binary data
- Needle.HayProfile contains haystack profiles (see [?])
- Needle.HayClusterType (not used so far) is intended to contain information used by clustering algorithms

Even though theoretically we would like to have only one Needles table containing all the needles in the Haystack, practically this is undesirable, since each type of needle contains a different type of data. If we put all the needles into one table, then the Data column would have to be declared of SQL type BLOB (binary object), which would mean that the database would treat the data as raw bits rather than typed objects, and we would not be able to perform data-type specific search (compare strings, etc.) Hence, we need to split the needles between several tables according to the type of data that they contain.

HayString

ID	Data
2730	Svetlana Shnitser
2735	Integrating Structural...
...	...

HayFloat

ID	Data
2759	0.98
5735	0.53
...	...

HayDate

ID	Data
1459	03-07-2000
3796	04-06-2000
...	...

Figure 4-3: Three Types of Needles Tables

Does that mean we have to define 9 separate Needle tables? Not really. First of all, the idea behind structured search is to search through the documents' metadata rather than their contents. It would make little sense to store the HayFiles and HayByteArrays, since those types of needles are used mainly to represent the files and their contents and there are no useful queries that can be applied to those objects. We can limit the list even further. For our search purposes, the URLs that are stored in HayURL needle are essentially strings, so we can put both HayStrings and HayURLs into one table. HayMIMEData contains a description of the mime type and encoding, so it is also a string. HayProfile and HayClusterType are the needle types that are used internally by Haystack, so they should not be of any interest to the user and there is no need to store them in the database at all.

Finally, we are left with only 3 different needles tables that we need to maintain — one containing Strings, another containing numerical values (Floats) , and a third one containing Dates. If we later decide that Haystack should include an additional type of data that does not fit into any of these tables, it should be relatively easy to incorporate another table into our query system.

Chapter 5

Querying the database

Now that we have defined the schema for the table, we will show how the user requests in our query format can be transformed into database queries. In theory, it would be interesting to find out whether or not we can postulate that it is possible to translate any query in the Lorel format into SQL given our data model. For now, however, we will illustrate the translation process with a set of examples.

5.1 Simple Queries

Suppose the user wants to see all documents written by Svetlana Shnitser. In terms of the data model, what we need to do is find all the straws that have a Tie.Author (all our tie labels begin with prefix "Tie.") pointing to a needle containing the string 'Svetlana Shnitser'.

The database query will look as follows:

```
SELECT Ties.FromID FROM Ties, HayString WHERE
        Ties.Label = 'Tie.Author' AND
        Ties.ToID = HayString.ID AND
        HayString.Data = 'Svetlana Shnitser'
```

For our example table in Figure 4.1, this will return {2730}.

Let's consider another example. Suppose the user wants to see all his phone bill statements since April of last year. According to our data model, he would need to search for a bale that has a title 'Phone bill', and a date greater than April 1999. First, just like we did in the previous example, we can select all straws that satisfy the first condition:

```
CREATE VIEW PHONEBILLS AS
        SELECTFromID FROM Ties, HayString WHERE
                Ties.Label = 'Tie.Title' AND
                Ties.ToID = HayString.ID AND
                HayString.Data = 'Phone bill'
```

Similarly, we can find all the documents that have a date of 04/1999 or later.

```
CREATE VIEW THIS_YEAR AS
        SELECT FromID FROM Ties, HayDate WHERE
                Ties.Label = 'Tie.Date' AND
                Ties.ToID = HayDate.ID AND
                HayDate.Data ≥ '04/01/1999'
```

Finally, we can find out which documents satisfy both conditions:

```
SELECT PHONEBILLS.FromID FROM PHONEBILLS, THIS_YEAR WHERE
        PHONEBILLS.FromID = THIS_YEAR.FromID
```

In this example, we used the views mainly to illustrate the process - in actuality, we can express this query in a single SQL statement. Since no row in our Ties table would satisfy both conditions at once, we need to work with two separate "copies" (aliases) of the Ties table (T1 and T2), so that we can pick the rows from one copy that match the title, pick the rows from the second copy that match the date, and then, again, select the ones that have the same FromID. The query would look as follows:

SELECT T1.FromID **FROM** Ties T1, T2, HayString, HayDate **WHERE**

T1.Label = 'Tie.Title' **AND**

T1.ToID = HayString.ID **AND**

HayString.Data = 'Phonebill' **AND**

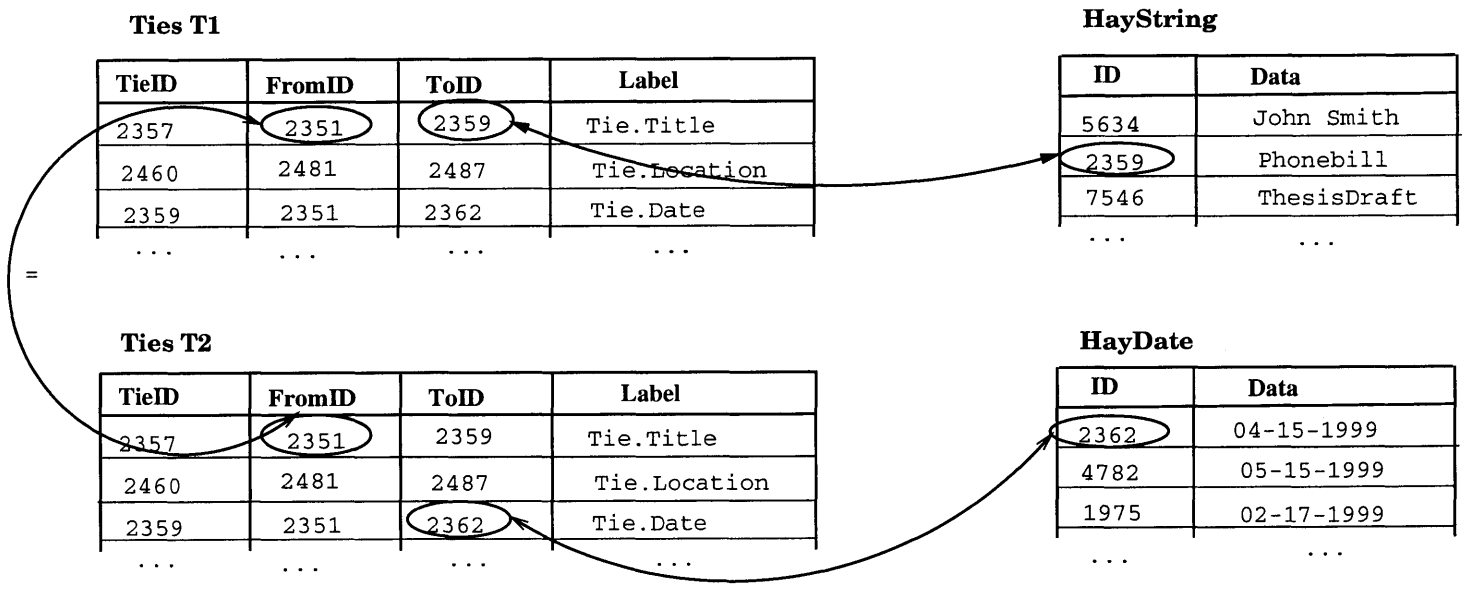
T2.Label = 'Tie.Date' **AND**

T2.ToID = HayDate.ID **AND**

HayDate.Data $\geq$ '04-01-1999' **AND**

T2.FromID = T1.FromID

Figure 5.1 illustrates how this query works with the tables:



Ties T1

TieID	FromID	ToID	Label
2357	2351	2359	Tie.Title
2460	2481	2487	Tie.Location
2359	2351	2362	Tie.Date
...	...	...	...

HayString

ID	Data
5634	John Smith
2359	Phonebill
7546	ThesisDraft
...	...

Ties T2

TieID	FromID	ToID	Label
2357	2351	2359	Tie.Title
2460	2481	2487	Tie.Location
2359	2351	2362	Tie.Date
...	...	...	...

HayDate

ID	Data
2362	04-15-1999
4782	05-15-1999
1975	02-17-1999
...	...

Figure 5-1: A Database Query Requiring Copies of the Tables

5.2 Multi-Layer Queries

5.2.1 Multi-Layer Constraints

In all our examples so far, we have considered the queries in which the constraints involved only one link between the document and the needle containing the desired value. As we stated in chapter 3, we would like for the user to be able to specify the constraints involving arbitrary numbers of links. Let's see what a query like that would look like given our schema.

This time, let's assume the user is looking for the documents in his haystack that mention the paper that he wrote earlier, entitled "MyPaper" for simplicity. In order to answer this query, we would again make use of the aliasing feature in SQL. We will use one copy of the Ties table (call it T1) to find the documents that have a "Tie.References" link. Then, using another copy (T2), we will find the document(s) the user had in mind — the ones that have a 'Tie.Title' pointing to a needle containing the string 'MyPaper.' Finally, we will select only those rows from T1 whose ToID matches a FromID in the rows we selected from T2. The query will look like this:

SELECT T1.FromID **FROM** Ties T1, Ties T2, HayString **WHERE**
    T1.Label = 'Tie.References' **AND**
    T2.Label = 'Tie.Title' **AND**
    T2.FromID = HayString.ID **AND**
    HayString.Data = 'MyPaper' **AND**
    T1.ToID = T2.FromID

Alternatively, we could express this query using subqueries instead of aliasing - the two statements are equivalent, and in this case the database query processor should convert the expression with subqueries into a join anyway.

SELECT FromID **FROM** Ties **WHERE**

```
Label = 'Tie.References' AND
ToID IN
(SELECT FromID FROM Ties, HayString WHERE
        Ties.Label = 'Tie.Title' AND
        Ties.ToID = HayString.ID AND
        HayString.Data = 'MyPaper' )
```

5.2.2 Multi-Layer Selections

Simiarly to the multi-layer constraints, we may wish to process multi-layer selections that we mentioned in Chapter 3 where the user is not looking for the document itself but rather for some straw to which there exists a path from the document. For example, the user could be searching for the names of all people who wrote a document about Haystack. Then, he would formulate his query as :

SELECT Doc.Author.Name **WHERE** Doc.Subject = 'Haystack'

In this case, it is the document that has a subject 'Haystack', but the user is not interested in the document itself but rather in the in the particular property of that document specified as Doc.Author.Name. To express this query in SQL, we can use aliases practically in the same fashion as we did for multi-layer queries. The only different would be that we will only need to examine the tie labels along the path and will not need to check for the values of the needle at the end of the path.

As a help to the user, if the query finds documents that match the constraints but do not have selected attribute, we may want to present those documents at the bottom of the screen with the appropriate error message and let the user decide whether or not those documents are relevant to his query.

5.3 Combining conditions

As we mentioned in our discussion of the user interface and query format, we do not allow arbitrary boolean conditions in the first incarnation of the structural query system. This was a purely practical decision. If our database schema was such that every straw occupied a single rown in the table, then we would not need to "understand" the query - translating arbitrary boolean query into SQL would merely involve replacing the attributes with the column names of our tables. However, for the reasons outlined in Chapter 4 we did not choose this design. With the schema that we chose, it is impossible to simply replace some strings with others in the query. In order to correctly process an arbitrary boolean combination of individual constraints, our query translator would need to parse it and infer the structure of the query expression to be able to determine how to correctly combine the results of the individual constraints. That is, if the query was of the form (A or B) and (C or D), we would need to know to take the union of the results of A and B, and C and D separately, and then intersect the two results sets. To make that possible, we would need to be able to parse boolean expressions.

Thus, for the remainder of this section, we will assume that we are only talking about conjunctions of individual conditions.

Translating a query with multiple conditions involves essentially the same SQL constructs as translating a query with multi-layer attributes. Once again, if we look at the schema for our tables (see Figure 4.1), we can see that every row of the Ties table corresponds to an edge in our data model. Therefore, we cannot find a row in the Ties table that would satisfy many constraints at once. Just as we did with multi-layer queries, we would need to use aliases to generate the copies of the table that we can work with, and then perform a join on those tables to find the straws that actually satisfy all conditions.

5.4 Determining Data Types

All along in this dicussion, we have focused mostly on specifying the attributes of the documents in the query and assumed that the system somehow "knows" which Needles table to look at for the values. Let's examine this issue more carefully. Suppose the user types in a query that asks for documents whose date is between 12/02/99 and 01/02/00. In order to process this request, first of all we need to identify the words in the query string that actually represent the values. Since we are allowing any standard SQL functions to be used in the 'where' clause and since we decided not to parse the query, we have to ask the user to delimit the actual values with quotation marks. Thus, the query may look as follows:

SELECT Doc **WHERE** Doc.Date **BETWEEN** "12/02/99" **AND** "01/02/00"

Second, we have to determine the types of the data that are used in the expression so that we can use the appropriate "Needles" table in our database query. One possible solution would be to rely on the mapping between tie labels and the type of data the ties point to (so for example, if the attribute is Tie.Name, then the value is a string.) However, in the design of the data model, it was assumed that tie labels are independent of the type of data that is stored in the needles at the endpoints of the tie.

One possible alternative would be not to enforce a strict correspondence between tie labels and data types, but to dynamically maintain a list of correspondences that we have seen so far. For example, we could notice that Tie.Author points either to a HayString needle, or to a person bale. Alternatively, we could store an additional field in the Ties table telling us which table, if any, contains the straw pointed to by each tie. But since we only have 3 types of needles in our database, we decided it would be much easier to simply try all 3 possibilities — for each possible data type, construct a corresponding Java object containing the value, and if succeed, look in the coresponding table. Even though this approach was simplest for the time being, it is not necessarily optimal — if we ever increase the number of different Needle tables that we need, we would unfortunately have to modify the code to try new possibilities.

Chapter 6

Persistent Storage and Updates to the Database

Up to this point in the discussion, we have assumed that the relational database already contains all of the necesary data from the user's Haystack. In this chapter, we will try to understand how we can guarantee this assumption. We will take a closer look at the issues of persistent storage in Haystack and will describe the major modifications that have been made to the storage mechanism. We will also discuss the issues of consistency between the data in the database and the persistent storage module.

6.1 Persistent Storage - old model

As of the summer of 1999, the model for persistent storage used the underlying file system and basically consisted of keeping a file for each straw in Haystack. A FileManager class was responsible keeping track of which directory the files should reside in and for providing input and output streams to the file given a HaystackID. HsPersistentObjectService included methods like `getObject(HaystackID id)` that would return a Straw given its ID and `register(Straw s)` that would write the straw out to a file.

HsPersistentObjectService also included a write-through cache in the form of a Hashtable mapping IDs to the Weak References to straws. WeakReference objects allow a program to maintain a reference to an object but do not prevent the object from being considered for reclamation by the garbage collector. They also allow a program to be notified some time after the collector has determined that an object has become eligible for reclamation. The reason that the cache of WeakReferences was introduced was the fact that in the absence of cache, if two services request the straw for the same ID, they will each have their own objects to work with, and neither will see the changes that the other made. If we maintain an object in the cache, then upon request, both services will get the same objects and changes made by both services will be visible. The cache stored WeakReferences rather than straws themselves because if it had stored straws themselves, it would be impossible to tell whether or not any service was still working with that straw and determine whether or not it is safe to evict that straw from the cache. With the WeakReferences, if no service has a pointer to a straw, that straw will be garbage-collected, and the WeakReference would be `null` signifying that removing this ID from the cache is safe.

Even though the old model was fairly simple, it had several fundamental flaws.

1. it was only designed to work with straws, so services that needed to maintain some persistent data that was not part of the data model had to devise their own means of storage.

2. any service could potentially change any straw, without making PersistentObjectService aware of the change

3. most importantly, it was not transactional.

In order for a system to be transactional, it has to satisfy 4 basic properties:

- Atomicity (each transaction is treated as a unit of operation, either all the transaction's actions are completed or none of them are.)

- Consistency (each transaction leaves the data in a consistent state)
- Isolation (each transaction sees consistent data at all times, i.e. if transaction A and B run at the same time, A's modifications (even if it commits while B is running) are not visible to B)
- Durability (once the transaction commits, its results are permanent and cannot be erased.)

As applied to Haystack, we need transactions to make sure that if a service does not run successfully, its "half-baked" results are not visible to the data model. For example, if the archiving process encounters a problem and never passes the document to the indexer to be indexed, the document will exist in the data model, but will never be returned in respose to a query. Thus, we want to make sure that either all of the steps happen — from creating the document bale to indexing the text — or none do and we are notified that we need to re-archive the document later.

6.2 The New Model

6.2.1 Persistent Hashtable

The centerpiece of the new model for persistent storage is the class PersistentHashtable. It is an abstraction that allows any service in Haystack to have persistent data even if it is not a part of the data graph. With this design, the data graph is merely a PersistentHashtable mapping HaystackIDs to straws maintained by the PersistentObjectService.

PersistentHashtable implements most of the methods of the regular java.util.Hashtable class (such as put, get, remove, keys, size), but it requires that all the objects it is working with be Serializable, because unlike the regular Java Hashtable, it uses Java serialization methods to save objects to disk. Similarly to the old implementation of PersistentObjectService, PersistentHashtable maintains a cache to ensure that there is only one copy of the object that is being worked on.

PersistentHashtable does not write the objects to disk itself. Instead, it relies on an object that implements the PersistentStorage interface to do that. It is crucial that PersistentStorage is an interface because at this point, we are still exploring different alternatives for persistent storage, and it is important to be able to switch painlessly between different implementations.

The current implementation of persistent storage uses the Sleepycat database, which supports the hashtable storage structure and is in theory transactonal [10]. All the data is stored in one single hashtable, so each instance of PersistentHashtable is in essence a "virtual" hashtable, containing a subset of the elements in the original hashtable. In order to guarantee that services do not overwrite each other's data, we make sure that if a key is unique within a PersistentHashtable, it is also unique within the real hashtable. To guarantee uniqueness, we append the service name and the hashtable name to each key (since a service may in principle have more than one hashtable with the same keys, and different services may have hashtables with the same names.)

We also want to guarantee that no service can overwrite other services' data. To that end, we created a KernelInterfaceObject that can only be accessed by the "kernel" services such as the HaystackRootServer, NameService, HsService, PersistentHashtable and a few others. When a "regular" service needs to create or access its Hashtable, it has to call the getPersistentHashtable method on its parent class HsServer and pass it the hashtable name. The HsServer, being a kernel service, will ask the KernelInterfaceObject to store the service name and table name in variable to which only KernelInterfaceObject has access. HsServer will then call the constructor of PersistentHashtable. PersistentHashtable, which also has access to the KernelInterfaceObject, will look up the service name and table name and clear the varaible. Then, it will use those names in generating unique keys to access the service's data.

For a more detailed discussion of kernel issues, see Appendix A.

6.2.2 Transactions and Service Queue

As we stated before, to ensure that no half-completed updates happen to the data model, we needed transactions. In particular, we deicided that when a service is triggered in respose to the change in the data model, all its actions should be considered a single transaction, so that either all the modifications carried out by that service are visible in the data model, or none are.

When a service runs, not only does it modify the data model, but it also may trigger an event that may cause another service to run. In the old model, if a service triggers an event but then Haystack is terminated, the event would not be "remembered", i.e. if the TypeGuesser has determined that the document is of type HTML, but the HTML service was not run because we terminated Haystack (or it crashed, for example), the HTML service would never be triggered again, and the system would never know about the contents of the HTML document that it archived.

Hence, we decided to implement a persistent service queue. It works as follows: whenever service A triggers service B, service B is added to the persistent queue as part of service A's transaction. At the same time, the dispatcher keeps looking at the queue, running services, and taking them off the queue when they are completed. Thus, if service A runs to completion, and then Haystack is shut down, service B will still be on the queue and will be run by the dispatcher the next time Haystack starts up. Note that a service is taken off the queue only after it successfully runs to completion, so if the system crashed in the middle of a transaction, the modifications to the data model would not be visible and the service would still be on the queue and would start all over when we restart Haystack.

Of course, the undesirable side effect of keeping a service queue is the potential of having a service that crashes Haystack every time it runs (hence it will never run to completion and will keep being triggered when Haystack is restarted). Thus, it would be a good idea in the future to provide a user command that would take a service off the queue.

6.2.3 Relational Database and Storage

We should note that the relational database is also one of the alternatives that we could use as a means of persistent storage (especially if it supports transactions.) This way, we could perhaps avoid duplicating the data for structural searches. However, it is important to keep the structured search and persistent storage as separate interfaces, regardless of how they are actually implemented.

6.3 Storing the Data in the RelDB for Searches

From the "storage" perspective, the relational database that we use for searches should support 2 functions:

1. It should be to rebuild itself from the persistent storage. Whether or not the relational database supports transactions, we still make an assumption that persistent storage is the only place where the data is stored consistently, so in case our system crashes, we need to be able to rebuild the database according to what's in the persistent store. To implement that functionality, we need to be able to detect failures and crashes.

   One way to do that, for example, would be write out a file upon closing haystack that would signify that no failures were encountered while the system was running. Also, if we catch an exception from the relational database, we should note in the file that the next time we restart, the relational database should be rebuilt from persistent storage. Currently, we do not try to detect failures but we do provide a rebuiltFromStorage() function in the RelDatabaseInterface that will be called if the user specifies -cleandb option on startup.

2. Incremental updates. Even though we know we can maintain the relational database by periodically rebuilding it from the transactional store, a more efficient solution seems to be to actually propagate the incremental changes to the relational database, and rebuild it from the transactional one only if we have a reason to believe that the data in the relational database is inconsistent or corrupted. Whenever the data graph is modified (e.g. by the user who archives new documents or issues a query, or internally by Haystack services), the modifications should be reflected in the database. To that end, the RelDatabaseInterface includes `putNeedle` and `putTie` methods. Even though the relational database is not required to be transactional, it would be nice if it were so, since that would decrease our chances of having to rebuild the database from the persistent store.

Chapter 7

Integrating Text Search, Database Queries, and Machine Learning

In this chapter, we will discuss how we can integrate the text search, the database seach, and the machine learning together. We will provide a quick overview of machine learning techniues and will present two ideas on how machine learning can be applied to both the text and the database queries.

7.1 High-level view

As we stated earlier, the original goal of Haystack was to allow the user to search through and navigate his corpus of knowledge. It was assumed that the primary search technique would be the keyword (text) search, and it was suggested that Lore [5, 1] would be used to perform semi-structured searches.

In order to provide the text search capabilities, a third-party text search engine (ISearch [11]) was used. The text of the documents was passed to it to be indexed, the query strings were propagated directly to the engine, and the matching documents and their scores, as returned by ISearch, were passed along to the user. In other words, the system was used as a black box.

As the work on Haystack progressed, not only did we decide to provide the structured search capabilities, but the idea was also proposed to incorporate machine learning functionality into our system. This raised an interesting and challenging question: how are we going to integrate the three systems (text search engine, database, and machine learning) together in a way that is coherent and takes full advantage of each subsystem's capabilities?

Before we proceed with the discussion, we need to provide an overview of what exactly do we mean by machine learning and which techniques were planned to be implemented in Haystack. These techniques were implemented as part of Wendy Chien's Masters Thesis, so see [3] for an in-depth treatment of the topic.

7.2 Quick Overview of Machine Learning Capabilities of Haystack

All information-retrieval algorithms represent documents as vectors where each dimension reflects presence or absence of a certain word in the document or the number of times the word occurs in that document. Then the similarity between two documents can be computed as the dot product between their word vectors.

According to the definition, a machine learning system is a system that uses sample data to generate an updated basis for improved performance on subsequent data from the same source and expresses the new basis in intelligible symbolic form. Put in simpler terms, a machine learning system in the context of an information retrieval system takes the user's feedback on a given query and uses it to produce improved results the next time the user inputs the same or similar query.

One technique that implements the above behavior is relevance feedback. It relies on the fact that users often input queries containing terms that do not match the terms used to index the majority of the relevant documents. Likewise, some of the relevant documents often contain a different set of terms than those in the query or in the most other relevant documents.

There are two approaches to this problem. One involves re-weighting the terms in the query based on the distribution of those terms in the documents that the user has marked before as relevant to the query. This may increase the score of the documents containing the terms with higher weight that otherwise may not have been retrieved. The other approach, called query expansion, relies on the assumption that terms closely related to the original query terms can be added to the query and thus retrieve more relevant documents. Unlike the previous approach, query expansion does not increase the ranks of the documents with re-weighted terms, but instead, helps find relevant documents that were not retrieved because their terms did not match the query terms [7].

A very well-known and simple query expansion algorithm that was implemented for Haystack was invented by Rocchio and is based on the following formula:

$Q_1 = Q_0 + 1/n_1 \sum_{i=1}^{n_1} R_i - 1/n_2 \sum_{i=1}^{n_2} S_i$ where

Q_0 is the vector containing the terms of the original query,

R_i is the vector for relevant document i,

S_i is the vector for nonrelevant document i,

n_1 is the number of relevant documents

n_2 is the number of nonrelevant documents.

Q_1 is therefore the vector sum of the original query plus the vectors of the relevant and minus the vector of nonrelevant documents.

7.3 Combining DB, IR, and ML

So how can we bring the database, the text search engine, and the machine learning unit together? Or going even further, suppose we have more than 2 kinds of search techniques — how do we built a system that intelligently combines their results and learns from the user's feedback? The most straightforward approach with current search techniques would be to integrate machine learning with the text search engine, since both make use of the word vectors for the documents being indexed, and treat the database as a "filter," i.e. if the user inputs a database query, our system should return only those documents among the results of the text search, that also match the database query.

Unfortunately, the first part of this plan is currently unfeasible since we are using a third-party search engine and can only treat it as a black box. Even though we plan to change this in the future, for now we have to assume that machine learning unit will be build on top of the search engine and will maintain its own set of word vectors.

But aside from practical considerations of whether or not we have an access to the source code of the search engine, there is a more interesting issue at hand. The proposed approach assumes that machine learning only operates on the text queries, as it was originally designed to do. Can we use the existing machine learning techniques to have our system learn from other types of queries (database queries in our case)? Or should there be a separate learning module for every kind of search that would use different techniques and "adapt" to the user in its own way instead?

Theoretically, it seems plausible that machine learning can be used with non-text queries. For example, our system could conclude that all of the relevant documents for a certain query have are dated this year, whereas all non-relevant were written last year or before, and hence, when the user asks the query again, the system would include the date in the query. Even though the abstract idea sounds simple, how one would implement such a system is not obvious.

7.4 First Design Alternative: Matching a Query is a Feature

All information-retrieval algorithms work in the "language-space." We can think of each word in the language as a "feature" that the document either does or does not have. If we try to draw the analogy in the metadata world, then the feature becomes the attribute-value pair, i.e. instead of representing a document in terms of words, we represent it in terms of all possible attribute-value pairs that may exist. So instead of asking "does this document contain this word?" we can ask "does this document have 'Svetlana' as an author?"

Thus, one approach that will allow the machine learner to work with database queries is to maintain a "feature vector" for each document that would contain both the language features of that document, and its database features (i.e. which queries it matched). Language features and database features will have different range of values, though — for the language features, the value would correspond to the number of times the word occurs in the document (perhaps scaled by the total times the word occurs in the corpus); for the database features, the value would be binary based on whether or not the document matches the given query (perhaps also scaled by how many documents match that query.)

7.4.1 Practical Limitations

In the language-space, the number of dimensions does not increase very fast, since number of words in the dictionary is somewhat limited and does not grow very quickly with the corpus size. Unfortunately, this is not the case with metadata because an attribute can potentially have an inifinite range of values, and hence we may end up with a space with an unlimited number of dimensions. How can we get around this problem? The simplest solution is to add dimensions "on demand", i.e. only when the user makes a query about a particular attribute and value. So if the user requests all documents with where Doc.References.Author = 'Svetlana', then for every document that matches the query, we add the coordinate corresponding to the query and set the value to 1. (We assume that the absence of the coordinate means that the query has not been executed yet.)

Our corpus is not fixed, however, and will grow as the user archives new documents and adds new links. Changing the corpus would immediately invalidate all the database features that we have computed so far. When we add new straws, we would not know which of the past queries they matched, and adding a tie between two straws may cause them to match the queries (and hence have the features) that they didn't match before.

Therefore, what we need to do is keep track of all past queries that have been issued and update our feature space as soon as our corpus changes. While this behavior would be desirable theoretically, practically this is unfeasible because rerunning all past queries and updating the feature vectors of all documents is a very resource-consuming task. As a compromise solution, we can update the feature vectors of matching documents at the time when the query is issued, schedule "complete" updates only once per certain time period, and accept the fact that our feature space may not always be completely up to date.

In order for machine learning techniques to be effective, the learner has to be trained. That is, to be able to classify the documents as relevant or nonrelevant for a given query, it needs to see some examples of the good and bad matches for this query if it has been issued before. If we update the corpus and rerun all queries and update the result sets, that would also invalidate the learner's training data (suppose before we had only 5 documents written on a certain date, and the ones that the user has chosen as relevant had "John Smith" as author. Now suppose the user has archived 20 more documents with the same date — the learner's data has all of a sudden become obsolete.) Therefore, not only do we have to periodically update the feature space, but we also have to retrain the learner on the new feature space.

7.5 Second Design Alternatvie: Multiple Spaces

Let's step back and look at the big picture of what we are trying to accomplish by storing the database features. What we would like to do is to enable the learner to distinguish which features are common among the documents marked relevant to the certain query by the user. If we treat each database query as a feature, then, in the example that we used before, unless the query "select Docs where Doc.Date $\geq$ 01/01/2000" has been made, the system would have no way to tell that the year is really the distinguishing feature among all relevant documents. In fact, even if we assume that the user has already made a date query for every day of this year, then when we add the vectors of all the relevant documents, we will end up with a few of them matching one date, a few matching the other, but neither one of those dates will have enough documents to be selected as relevant to the query. In other words, our system will have a lot of data points but will not be able to deduce anything from them.

So it becomes apparent that we need a separate way to compute similarity for each different data type, that is, we need a "date-space", a "number-space", a "string-space" and so on. In each space, instead of recording 1 if a document has a particular attribute-value pair and 0 otherwise, we can instead record the value of the attribute itself if this attribute exists in a document. Then, when doing relevance-feedback or query expansion, we can ask each space to tell the learner which features it thinks are common among the relevant documents to this query. In language-space, this functionality be implemented as Rocchio's algorithm [6], for example. In other spaces, we can devise a different heuristic that will determine the value based on how close the dates or strings are correlated.

If we could indeed devise the right method for determining correlations between attributes, then the multiple-space approach would turn out to actually be more space-efficient, since instead of storing a feature for each attribute-value pair, we will only have one feature per attribute. However, if we are to resort to existing machine-learning methods, this approach cannot be used. Mchine learning algorithms are monotonic. In a sense, they find a threshold that divides "good" and "bad" (or relevant and nonrelevant) documents based on some characteristics. If we record the actual values of the attributes as features, we would need to define not a threshold but a range of values that are "good." For example, if the query is for all documents dated on 07/05/99, then related matches would fall the in range between 07/03/99 and 07/07/99.

To use current machine learning algorithms with database queries, we may again have to resort to storing a feature for each query. But unlike the previous approach, for each query of the form "select doc where doc.a.b.c = X", instead of recording the value corresponding to doc.a.b.c we would record how simiar doc.a.b.c for this document is to X (if the document has an attribute doc.a.b.c, of course.) For instance, we could use edit distance to compare strings, and simply subtract dates. Again, this approach would mean maintaining a feature for each query, but the information would be more meaningful than simply a bit stating whether or not the document matches the query and would make relevance feedback on database queries more efficient.

Chapter 8

Query System Architecture

Now that we have discussed the design of the individual modules, we will put together the big picture of how the query process actually works in Haystack.

8.1 The Old Query System

In the absence of structural search and machine learning modules, querying was handled by the HsQuery service that implemented the HsQuery interface. In short, HsQuery took the query string, passed it to the underlying text search engine, created the query bale to which it attached all the matching documents, and returned the ID of the new bale back to the user interface. If the query string exactly matched a previously entered query, the new results were added to the already-existing query bale for that query. Figure 8.1 shows a sample query bale.

HsQuery also included methods like `merge` that merged the results of two queries, and `queryWithin` that issued the new query against the results of the previous query. Additionally, there was a QueryUtils class, which was essentially a library encapsulating several methods frequently used by HsQuery, such as building the query bale or populating it with the matching documents.

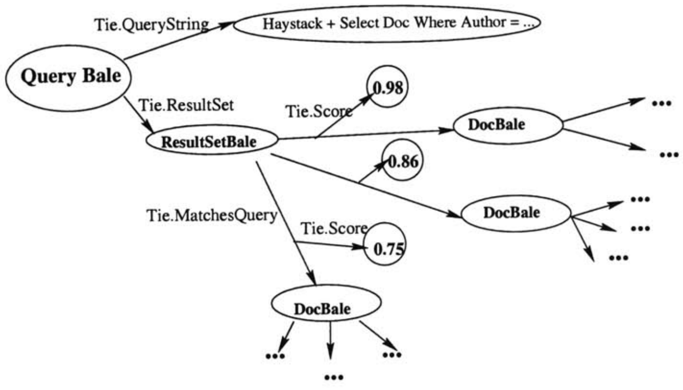


Figure 8-1: The Query Bale

8.2 The New Design

As we stated before, the goal of Haystack is to be the system that acts as an interface to different types of search engines by dispatching the user's query to the appropriate modules and intelligently combining their results. This is the goal that we tried to achieve with our new design of the query system. What was previously its primary task (the text search) has now become one of the tools used by the new query module.

In the old design, the information flow was simple and clear: HsQuery passed the query string to the IR system, the IR system returned ranked results, HsQuery built a query bale and returned the ID of the bale to the user. With the appearance of structured query and machine learning, the information flow has become a lot more complicated, and we decided to delegate the control flow to a separate object called the QueryManager. The QueryManager has several "subordinates" (see Figure [?]).

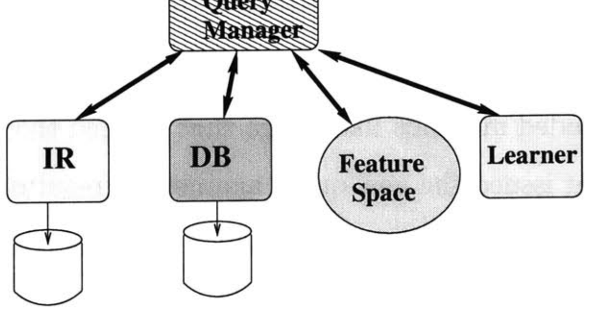


Figure 8-2: Different Modules of the Query System

The text search engine (IR) is the same unit that we previously used for text queries. The relational database querying system (DB) parses the semistructured part of the query, translates it into SQL and passes it to the actual database (as we discussed in detail in the previous chapters). The Learner implements machine-learning algorithms. The FeatureSpace is the unit responsible for maintaining the different features we discussed in Chapter 8 (to be used by machine learning.) Finally, the Aggregator is an object that knows how to combine the results of individual subsystems.

Here are two scenarios illustrating how QueryManager works:

Scenario A. The query string is "new" (has never been issued before)

1. Create the new query bale

2. Issue the text portion of the query (if such exists) to the IR system, call the result set R1

3. Issue the database portion of the query (if it exists) to the DB system, call the result set R2

4. Update the feature space with R2

5. Check if R1 contains any old query bales. For every query bale that it contains, compute its feature space as a sum of the features of the documents in its result set. Select n documents whose feature space is the most similar to the feature space of R1 (call the resulting set Q)

6. Call Aggregator with R1, R2, and Q

7. Populate the query bale with the ranked results returned by the Aggregator and return the HaystackID of the bale to the user.

Scenario B. The query has been executed before (the query string is similar to some "old" query)

1. Work with the old query bale

2-5. same as in Scenario A

6. call the Learner to perform queryExpansion.

7. Pass the textual part of the expanded query to the IR (call the result set R3), and the database part to the DB (call the result set R4)
8. Call the Aggregator with (R1, R2, R3, R4, and Q)
9. same as step 7 in scenario A.

Even though we decided that QueryManager should be a separate class, currently, the "control" procedure that we outlined above is still part of HsQuery. In order to incorporate the database search into HsQuery, we also needed to add methods like `structQuery()`. Similarly to the original `query()` method, `structQuery()` passes the query to the database and does some initial postprocessing of results (if the database query contains multiple constraints, it assigns scores to matching documents based on how many individual constraints they satisfy, as we discussed in Chapter 3). The reader is welcome to look through Appendix B for the interface definitions of HsQuery.

8.3 The Aggregator

As me mentioned above, the aggregator module is responsible for taking the results produced by the database search, the text search, and the text search with expanded query, and producing the final ranking. Currently, it assigns relative importance to the different "opinions" (the result sets that are passed to it) by scaling each set of scores by a certain factor. For example, database scores may be scaled by 0.7, while expanded query results may be weighted by 1.5. In the current implementation of the aggregator, the weights are static and were derived empirically. In the future, we could let "advanced" users tweak the weights. An even bigger enhancement would be to make Aggregator learn from the user interaction and dynamically adjust the weights.

8.4 The DB Module

8.4.1 RelDatabaseInterface

The DB module that we referred to above implements most of the structural search functionality that we outlined in the previous chapters. The functions are defined in the `RelDatabaseInterface`, and implemented by the class `RelDatabase`.
Recall that our structural search unit should support several types of functions:

a) it should be able to rebuild the database from persistent storage
b) it should be able to incrementally update the database when the user or the system changes the data graph
c) it should be able to return a set of matching documents given the query in the format that we outlined in Chapter 4.
d) it should provide a list of labels and possible values given a path through the data model (for DataGuides)

The complete interface definition and the details of each method are presented in Appendix B. Here, we will outline several key methods.

- `RebuildFromStorage()`, as the name suggests, rebuilds the database from scratch based on the data contained in the persistent storage

- `putNeedle()` method inserts the data contained in the needle into the appropriate table in the database (see Chapter 4 for discussion of needles.)

- `putTie()` makes a record of the new edge in the Ties table

- `query()` translates the query from our input format into SQL, executes it and returns the set of matching documents.

- `guideLabels()` is the method that should be used by the dataguides to return a set of labels that can complete a given path.

8.4.2 Connecting to the Database

In order to implement structural searches, we needed to be able access the relational database from Java code. Furthermore, we wanted to make our design modular enough so that it could work with any relational database that supports the set of features we need. The JDBC API seemed to be the ideal solution for this problem. The JDBC API allows an application to access virtually any database on any platfrom with a Java VM, as long as the driver for that database implements the JDBC API.

Working with JDBC API is fairly simple and usually involves 3 basic steps: connecting to the database, creating and executing a statement, and processing the result set. Aside from connection, there is virtually no dependence on the database itself. The majority of the code involves working with the JDBC Statement class, which takes as an argument a string representing the SQL statement. The results of query are obtained through the ResultSet class, which allows us to iterate through results. See [12] for a detailed description of the JDBC API.

8.5 Timing and synchronization

In our current implementation, the queries happen sequentially - first we query the IR system, then the database, then expand the query and run it against IR and DB again. An interesting design alternative would be to allow different query subsystems to answer the queries asynchronously. That is, instead of querying IR and DB sequentially, we could issue the queries in parallel, then wait for both results to come back. This approach may bring a significant improvement in performance, especially if we choose to have several different search engines.

Chapter 9

Future Work

In the course of this project, we have developed the infrastructure for database searches and have implemented basic search functionality. In this chapter, we will discuss the ways in which future developers of Haystack can enhance the structural query system.

9.1 Integrating Machine Learning and DB Queries

In Chapter 8, we have proposed a new design for using machine learning techniques on structural queries. We described how we can learn from database queries by defining several "spaces" based on different data types that would allow us to compute the extent to which a document is relevant to a query.

Unfortunately, due to the time constraints, we did not implement this functionality. It would be very interesting to experiment with this approach and see to what extent our system can learn from database queries and how that could improve our search results.

9.2 String Matching

In the current incarnation of our structural search system, if the user types a query involving an attribute whose values is a string, we perform exact match of the value that the user has typed in against the values in the database. However, it often happens, especially with names, that there are multiple ways that the name may appear in the documents. For example, the author of the document may be listed as A.B. Smith, or Andrew Brian Smith, or Andrew B. Smith, and all three names would refer to the same person.

This problem has been addressed by the CiteSeer project [13] which aims to provide indexing and retrieval of scientific literature. In publications or research papers, the appearance of names is especially important, since the users would often search for papers by a given author. It seems like this feature could also be useful in Haystack — it would make our system more intelligent and helpful to the user.

9.3 Adding features to our query language

As we mentioned in Chapter 4, in our initial version of the structural query system, we chose to implement a subset of the features available in Lore. It would need to be determined whether or not it is possible to translate an arbitary query in Lorel into SQL given our data model, and if so, it would be desirable to implement the rest of the features, such as arbitrary boolean combination of constraints, wild cards in expressions, and range variables.

A smaller-scale project would be to add the dataguide front-end to our user interface. Currently, we provide the back-end functions that return the list of labels that can complete a given path through the data model. It would be convenient if the user could select one of the lables we returned, and have it appended to the query string that he is trying to formulate.

9.4 Issues of Persistent Storage

Another issue that should be paid attention to is persistent storage and consistency of information in the store and in the relational database. Even though there is a method for rebuilding the database from persistent store, currently it can only be called by specifying `-cleandb` argument to Haystack. What we need is a way to detect if there is a reason to rebuild the database from storage (i.e. if there was a crash or a problem with the database), halt all updates to persistent store and the database, and then populate the database with the values contained in the persistent store.

9.5 Refining the Data Model

If we recall our data model that we described in Chapter 2, we can see that virtually all information in Haystack is stored there. The side-effect of this design is that when browsing his Haystack, the user may come across the auxiliary information that in theory, he should not even be aware of. One possible solution to this approach that we would like to have implemented in the future is to define several "views" of the data model using Java interfaces. For example, all straws that can be presented to the user would be marked as Visible, whereas the straws that store auxiliary information used by services should be marked Internal and should not be displayed to the user by the user interface.

9.6 Indexable Interface

In our data model for the relational database, we have no notion of bales — we merely record the graph structure in the Ties table and store the contents of certain needles in several Needles tables. Therefore, the database query can return arbitrary straws in the results set - they may be needles, or they may be bales - as far as the database is concerned, it makes no difference so long as both have HaystackIDs.

This is not the case with text search. Text search was developed with the assumption that the user is only interested in documents and is not interested in seeing separate Date or Subject needles. Moreover, when a document is indexed, the text of the neighboring needles is added to the text of the document, so that for example, if the the 'Author' tie pointed to a needle containing 'Svetlana', then the document itself would also match the text search on "Svetlana".

This creates an inconsistency between text search and the database search. For example, this way needles will only have the features corresponding to the queries that they matched, but bales have a feature for every word that the document contains, plus the words of the neighboring needles, and on top of that features for the database queries. It would be nice if this issue could be straightened out - perhaps that would involve creating the Indexable interface and deciding which pieces of the data model should be indexed and which should be returned as the reuslts of the text and database queries.

Chapter 10

Conclusion

In the course of this project, we have designed and implemented a system for performing structural searches in Haystack.

The information stored in Haystack is semistructured, i.e. the data objects have certain structure, but it is not guaranteed to be the same across all types of objects handled by Haystack, and can change dynamically as more objects are being archived. The challenge of this project was to develop structural search system that performs database-like queries on semistructured data using current relational database technologies.

To achieve this goal, we have designed a database schema that would allow us to store our data model in a relational database without imposing any restrictions on it. We have designed the user interface and have specified the format in which the user can enter database queries and implemented procedures that translate user queries into SQL that we in turn pass to the database. Since our query language assumes familiarity with the data model, we have provided a tool called DataGuide that would allow the user to explore and deduce the model of his Haystack.

The original goal of Haystack is to be a personalized information retrieval tool that dispatches user queries to different types of search engines and combines their results. Haystack should also be able to learn and adapt to individual user's needs by noting which documents the user considers relevant. However, most machine learning methods were designed with only the text search in mind. In this thesis, we have built a system that intellingently integrates text search and database search, and have proposed new ways in which machine learning algorithms can be applied to the results of structural search.

We hope that future developers of Haystack will pursue further improvements to our structural search unit and will explore our ideas of learning from database queries. We also hope that Haystack will become an indispensable tool for not getting lost in the information space.

Appendix A

"Kernelification" of Haystack

This appendix will describe the modifications to Haystack that were carried out in the summer of 1999 as part of the "kernelification" process. This discussion is somewhat tangential to the main topic of this thesis, but it should be of interest to future developers of Haystack.

A.1 What is Kernelification

Earlier in this thesis (Chapter 2) we mentioned that Haystack data model contans not only of the contents of the archived documents, but also auxiliary information. Every service is represented by its own bale, and every straw that the serice creates has a 'Creator' tie pointing back to the service bale. One of the reasons for maintaining this information was to be able to reverse all the changes made by a service that is buggy or gone haywire.

In the original implementation of Haystack, every service was aware of its own HaystackID, its name, and its package name, and was passing this information to ObjectCreatorService whenever it was creating a new straw. Absolutely nothing prevented a buggy or a malicious service to pass another service's HaystackID in the call to ObjectCreatorService and take no responsibility for creating the straws. Likewise, for every service, we needed to store the transaction that it was part of, but this information clearly should not have been made available to the service itself.

The point of Haystack kernelification was to separate services into "kernel-level" and user-level (or service-level), as it is commonly done in operating systems, so that system-specific information necessary for tracking and transaction management was only available to functions known and trusted to the kernel.

The services that became part of the kernel are the ones that in essence comprise the "operating system" of Haystack. Those include `HaystackRootServer`, `HsNameService`, `HsDispatcher`, `HsCounter`, `HsObjectCreator`, `PersistentHashtable`, and `TransactionManager`. `HsService` (the parent class of all services) and `Straw` were also made part of the kernel. `HsService` contains service-specific information (its HaystackID, name, and packagename) as private variables which as not accessible by its subclasses. `Straw` stores its own ID and its creator as private variables as well.

A.2 ThreadLocals

Even though we could store the ID of the service as a private variable in HsService, we could not do the same with transactionIDs since our application is multi-threaded and the same service could be running as part of several transactions (we normally have only one instance of each service.) Thus, it would be desirable if we could store the information in the thread iself. Java 1.2 defines `ThreadLocal` variables that allows us to do that.

ThreadLocal objects are typically private static variables in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID). ThreadLocal variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable. Thus, our kernel services can set the values of the ThreadLocal variables before they invoke the service, and then look up those values when the service makes calls to the kernel again (e.g. calls `ObjectCreatorService` or `PersistentHashtable`.)

A.3 `KernelInterface` and `KernelInterfaceObject`

We decided to declare the ThreadLocal variables in the HaystackRootServer and to create a `KernelInterface` that will provide the methods for kernel services to get and set the properties kept in the ThreadLocals, such as TransactionID and ServiceID. We also defined a `KernelInterfaceObject`, a private inner class of `HaystackRootServer`, that could access those variables and implemented `KernelInterface`.

We needed to ensure that only kernel services had access to ThreadLocal variables defined in `HaystackRootServer`. For that, in every kernel-level service, we defined `setupKernel(KernelInterfaceObject kio)` method and upon initialization, passed the KernelInterfaceObject as a "ticket" to being admitted to the kernel level. Enclosed at the end of this appendix is a definition of KernelInterface and a snippet of code from HaystackRootServer that should illustrate our design.

A.4 Examples

In section 6.2.1 we described how KernelInterfaceObject is used to retrieve the ServiceID from the ThreadLocal variable by PersistentHashtable. In essence, it works as follows. Recall that every service is a subclass of HsService. In order to access a PersistentHashtble, a service calls HsService's `getPersistentHashtable()` method. HsService asks KernelInterface to set the ServiceID property to the ID of the service that this instance of HsService represents and calls the constructor of PersistentHashtable. PersistentHashtable, itself being a kernel service, asks KernelInterfaceObject to look up the value of the ServiceID in the ThreadLocal variable and uses it to "wrap" all the keys (see Section 6.2.1).

Similarly, when HsDispatcher is about to invoke a service, it starts a new transaction. To record the TransactionID, it asks its KernelInterfaceObject to set the corresponding property in the ThreadLocal variable. As the service runs, it makes calls to the PersistentObjectService (to create new needles, for example) which in turn calls persistent storage. Persistent storage that, of course, is also part of the kernel, looks up the TransactionID stored in the thread and makes an update to the data graph as part of that transaction.

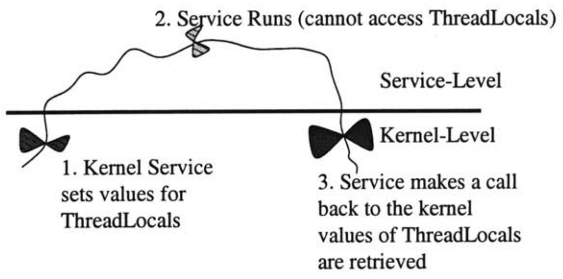

2. Service Runs (cannot access ThreadLocals)
Service-Level
Kernel-Level
1. Kernel Service sets values for ThreadLocals
3. Service makes a call back to the kernel values of ThreadLocals are retrieved

Figure A-1: Use of ThreadLocal variables by the kernel services

A.4.1 Code Snippet of HaystackRootServer

```
public class HaystackRootServer {

  // Cached copy; pass to everyone
  private KernelInterface kio = new Kernel();
  // Property table
  private Hashtable threadPropertyTable = new Hashtable();

  // Kernel object
  private class Kernel implements KernelInterface {

    private String[] defaultKeys = new String[] {"ServiceID",
                                                 "Event",
                                                 "TransactionID"};
    // Constructor
    public Kernel() {
      for (int i = 0; i < defaultKeys.size; i++)
        threadPropertyTable.put(defaultKeys[i], new InheritableThreadLocal());
    }

    // Retrieves reference to thread-specific property
    public Object getProperty(String key)
```

```
      throws KeyNotFoundException { ... }

    // Sets thread-specific property
    public void setProperty(String key, Object value)
      throws KeyNotFoundException { ... }

    // Creates new thread-specific property
    public void newProperty(String key) { ... }

    // Creates new thread-specific property with initial value
    public void newProperty(String key, Object initialValue) { ... }
  }

  // Init method
  public void init() {
    HsDispatcher.setupKernel(kio);
    ...
  }
}
```

A.4.2 KernelInterface

```
public interface KernelInterface {

  // Retrieves reference to thread-specific property
  public Object getProperty(String key) throws KeyNotFoundException;

  // Sets thread-specific property
  public void setProperty(String key, Object value)
    throws KeyNotFoundException;

  // Creates new thread-specific property
  public void newProperty(String key);

  // Creates new thread-specific property with initial value
  public void newProperty(String key, Object initialValue);
}
```

Appendix B

Interface Definitions

B.1 RelDatabaseInterface

```
public interface RelDatabaseInterface {

  /** Initializes the database assuming the data is in a consistent state
   *  or Haystack is starting up clean.
   */
  public void init() throws HsServiceInitException;

  /** Initializes the database assuming that some error has occured
   * in the persistent storage. Creates new tables and populates them
   * with the data from persistent storage.
   */
  public void recover() throws HsServiceInitException;

  /** Populate the databases with the data from persistent storage
   *  (assume there was a crash)
   */
  public void rebuildFromStorage();

  /** Enter the needle into the database
   * param ID - HaystackID of the needle to put in
   * param label - the label of the needle
   * param Data - the data contained in that needle
   */

  public void putNeedle(HaystackID ID, String label, Object data);
```

```
/** Enter the tie into the Ties database
 * param from - the Straw that the tie to archive points from
 * param to - the Straw that the tie to archive points to
 * param tie - the Tie to archive
 */

public void archiveTie(Straw from, Straw to, Tie tie);

/** Updates the info about the straw in the database
 * param s - the straw whose info should be updated
 * note: theoretically we only add things to haystack, not
 * remove anything, but since Straw has a detachLink method,
 * we have to implement a method that updates a straw in the database
 * as well.
 */

public void update(Straw s);

/** Translates the query into SQL, issues it, and retuns
 * a set of documents matching the query
 * param args - the query to issue (contains only one condition)
 * return a ranked set of documents that match the query (all ranked
 * equally in this case since the input query has only one condition
 * and all documents matching it have a score of 1)
 */
public RankedHaystackIDs query(DBQuery args) ;

/** Returns a set of labels that can complete a given path
 * param path - the path to return the labels for
 * exception InvalidPathException thrown if the path does not
 * exist in the data graph
 */
public Vector guideLabels(String path) throws InvalidPathException;

/** Returns a set of values that can be found
 * by traversing the given path through the data model
 * param path - the path to return the values for
 * exception InvalidPathException thrown if the path does not
 * exist in the data graph
 */
public Vector guideValues(String path) throws InvalidPathException;

/** Returns the set of all unique labels currently stored in the database
 * This is an auxiliary function used to make our user interface
 * case-insensitive with respect to the tie labels.
```

```
   * (since the database may be case-sensitive)
   */
  public Hashtable getLabels();

  /** Closes the connection to the relational database
   */
  public void close();

}
```

B.2 HsQuery

```
/**
 * This is the interface for the querying service.  This service
 * will respond to query requests and return the result as a
 * QueryResultSet
 *
 */
public interface HsQuery {

  /**
   * the name of this service
   */
  public static final ServiceName name =
              new ServiceName("HsQuery","$Revision: 1.9 $");

  /**
   * performs the query, creating a query document consisting of
   * the query result set and the query string used.
   * param queryString  - the string of terms to query with
   * return the HaystackID of the QueryDocument that
   * was created.
   * exception QueryException if there is a problem with the query
   */
  public HaystackID query(String queryString) throws QueryException;


    /* Same method as above, with an additional parameter
     * that is passed in by the web GUI to store new results of the query
     */
    public HaystackID query(String queryString, Vector newResults)
    throws QueryException;
```

```
/**
 * performs the query, creating a query document consisting of
 * the query result set and the query string used.
 * param queryString -  the string of terms to query with, to be passed
 * to the text search engine
 * param DBQueries -  a vector of constraints specifying the db query,
 * to be passed to the relational database
 * param newResults  - an additional parameter passed by the GUI to
 * store new results
 * return the <tt>HaystackID</tt> of the <tt>QueryDocument</tt> that
 * was created.
 * exception QueryException if there is a problem with the query
 */
 public HaystackID query(String queryString, Vector DBQueries,
                         Vector newResults) throws QueryException;

 /** return a list of possible labels that can be appended to
  * the input string to form a valid path through the data model.
  * exception InvalidPathException is thrown if the path does not
  * start with "Bale."
  * param path - the path to complete
  */
 public Vector guide(String path)
     throws InvalidPathException, QueryException;


/**
 * merges two queries. The resulting Bale will hold the set
 * of results that was in both queries. Each of the merged queries
 * points to the merged Bale with a "instanceOf" tie.
 * param First, Second the two Bales to merge
 * param newResults a vector that will be used by the web gui
 * return the HaystackID of the merged ueryDocument that
 * was created.
 * exception QueryException if there is a problem with the query
 */
public HaystackID merge(Straw First, Straw Second, Vector newResults) throws
 QueryException;
/**
 * Updates an existing query bale, adding to the bale any new items
 * that match the query string.
 *
 * param queryDocID the haystackID of the query bale to update
 * exception QueryException if there is a problem with the query
 */
```

```
    public void update(HaystackID queryDocID) throws QueryException;

  /**
   * Performs the query, creating a query document consisting of
   * the query result set intersected with another query result set,
   * storing also the query string used and a tie to the query set
   * intersected with.
   *
   * param queryString the string of terms to query with
   * param intersectBale the query result bale to intersect with
   * return the HaystackID of the QueryDocument that
   * was created.
   * exception QueryException if there is a problem with the query
   */
  public HaystackID queryWithin(String queryString,
                                Bale intersectBale) throws QueryException;

  /**
   * Performs the query, creating a query document consisting of
   * the query result set unioned with the addition set and
   * after subtracting out the removals.
   *
   * param queryString the string of terms to query with
   * param additions a vector of HaystackIDs to add
   * param removals a vector of HaystackIDs to remove
   * return the HaystackID of the QueryDocument that
   * was created.
   * exception QueryException if there is a problem with the query
   */
  public HaystackID queryRefine(String queryString,
                                Vector additions,
                                Vector removals) throws QueryException;


}
```

B.3 PersistentHashtable

```
/** This class provides persistent storage in the form of a hash
 * table.  All keys and values for the hash table must be
 * serializable. Maintains a primitive form of cache.
 */
public class PersistentHashtable {

  /** The cache that holds the objects that have recently been accessed.
   */
  private Hashtable cache;

  /** Pointer to the persistent storage */
  private PersistentStorage storage;

  /** Pointer to the object that allows access to ThreadLocal Variables*/
  private static KernelInterface kernel;

  /**
   * Establishes an interface to the kernel.  Called by
   * HaystackRootServer, which grants this class
   * special access to kernel-protected data.
   *
   * param  ki  the object that stores the interface to the kernel
   */
  public static void setupKernel(KernelInterface ki) {...}

  /**
   * Constructs a new persistent hash table with the identifier
   * stored in a thread-local variable as the owner of the hash table.
   *
   * exception  SecurityException  if the hashtable identifier is
   *            not specified in the "HashtableOwner" property
   */
  public PersistentHashtable() {...}

  /**
   * Puts the specified key-value pair into the hashtable.
   *
   * param      key    the key to put
   * param      value  the value associated with that key
   * exception  SecurityException  if the value of the transactionID
   *            corresponding to this operation cannot be retrieved from the
   *            threadLocal
   */
  public void put(Serializable key, Serializable value) {...}

  /**
   * Removes the key-value pair associated with the given key.
   *
```

```
    * param       key  the key of the pair to remove.
    * return      the deleted value, or <code>null</code> if the key
    *             is not in the database.
    * exception   SecurityException  if the transactionID associated
    *             with this operation cannot be retrieved from the
    *             threadLocal
    */
   public Object remove(Serializable key) {...}

   /**
    * Retrieves the value associated with the given key.
    *
    * param       key  the key for which the value is retrieved
    * return      the value associated with the key, or <code>null</code> if
    *             the key is not found
    * exception   SecurityException  if the value of the transactionID
    *             associated with the current operation cannot be retrieved
    *             from threadLocal
    */
   public Object get(Serializable key) {...}

     /**
      * Tests whether the hash table contains the specified key.
      *
      * param   key  the key to look for in the hash table
      * return  <code>true</code> if the key is in the hash table;
      *         <code>false</code> otherwise
      */
   public boolean containsKey(Serializable key) {...}

   /** Removes the Straws which are no longer used by any services from the cache
    */
    public void cleanCache() {...}

   /** Returns an enumeration of keys in this hashtable*/
   public Enumeration keys() {...}

   /** Returns the number of keys in this hashtable */
   public int size() {...}
}
```

Bibliography

[1] E. Adar. Hybrid-Search and Storage of Semi-structured Information. Master's Thesis, Massachusetts Institute ofTechnology, Department of Electrical Engineering and Computer Science, May 1998.

[2] I. Lisansky. A Data Model for the Haystack Document Management System. Master's Thesis, Massachusetts Institute ofTechnology, Department of Electrical Engineering and Computer Science, February 1999.

[3] W. Chien. Learning Query Behavior in the Haystack System. Master's Thesis, Massachusetts Institute ofTechnology, Department of Electrical Engineering and Computer Science, May 2000.

[4] R. Goldman, J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. Proceedings of the 23rd VLDB COnference Athens, Greece 1997

[5] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom. Lore: A Database Management System for Semistructured Data
`http://www-db.stanford.edu/lore`

[6] J. Rocchio. Relevance Feedback in information retrieval. In *The SMART Retrieval System — Experiments in Atomatic Document Processing*, pp. 313-323

[7] W. Frakes, R. Baeza-Yates. Information Retrieval. Prentice Hall, 1992

[8] Introduction to Structured Query Language.
`http://w3.one.net/ jhoffman/sqltut.htm`

[9] B. Rhodes, T. Starner. Remembrance Agent. A Continuously Running Automated Information Retrieval System. The Proceedings of The First International Conference on The Practical Application Of Intelligent Agents and Multi Agent Technology (PAAM '96), pp. 487-495.

[10] Sleepycat Software. `http://www.sleepycat.com`

[11] ISearch Text Search Engine. `http://www.cnidr.org/ir/isearch.html`

[12] JDBC API Tutorial and Reference, Second Edition Addison-Wesley, 1999

[13] S. Lawrence, K. Bollacker, C. Lee Files. Indexing and Retrieval of Scientific Literature English International Conference on Information and Knowledge Managements, pp. 139-146, 1999

[14] D. Karger, L. Stein. Haystack: Per-User Information Environments `http://haystack.lcs.mit.edu/papers/`

[15] MyYahoo. `http://www.myyahoo.com/`

[16] DirectHit Search Engine. `http://www.directhit.com/`

[17] PurpleYogi softare. `http://www.purpleyogi.com/`

[illegible]