A Data Model for the Haystack Document Management System

by

Ilya Lisanskiy

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1999

Author .....
Department of Electrical Engineering and Computer Science
February 8, 1999

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

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

A Data Model for the Haystack Document Management System

by

Ilya Lisanskiy

Submitted to the Department of Electrical Engineering and Computer Science
on February 8, 1999, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Haystack is a novel personal information management system whose goal is to provide an intuitive interface to a user's documents. This thesis describes the author's efforts to advance the system in several directions. First, we analyze the problem of metadata representation and present a data model based on a directed graph structure. The data model is highly flexible in expressing relationships among data. In the course of this thesis we attempt to define a document and offer our vision on this subject. Second, this thesis describes a substantial redesign of the Haystack system. In particular, we describe the new implementation of the services involved in the archiving process. Finally, we describe an implementation of a tool that enables the creation of word-frequency profiles that the Haystack system can use to adapt to the user.

Thesis Supervisor: David R. Karger
Title: Associate Professor

Acknowledgments

I would like to thank, first and foremost, David Karger, my thesis supervisor. During the past few months, David has been a great source of guidance and I have come to appreciate his abundance of ideas, optimism and unintrusive management style.

Haystack has been a team effort and I am indebted to all the people who have contributed in some way to the Haystack project: Professors David Karger and Lynn Stein, and students Eytan Adar, Mark Asdoorian, Yana Ageeva, Damon Mosk-Aoyama, Aidan Low, Christina Chu, Eric Prebys, Orion Richardson, Jing Qian and others. I owe extra thanks to Damon with whom I worked closely and who contributed much time to the work presented in this thesis.

I would also like to express gratitude to my parents, my sister Asya, and the rest of my family, for their love, guidance and support.

I thank my friends Boris Raykin, Mike Bryzek and Dani Katz for proofreading my thesis. Finally, I would like to thank Natasha Skorodinsky whose fun company, along with the company of the three people above, made the past few months an exciting and enjoyable experience.

Contents

1 Introduction **13**

1.1 Haystack Project 13

1.2 Implementation 14

1.3 Goals of This Project 14

1.4 About This Thesis 15

2 Background and Related Work **17**

2.1 Background 17

2.1.1 Information Retrieval 17

2.1.2 Metadata and Its Use in Haystack 18

2.2 Related Work 19

2.2.1 Metadata Representation 19

2.2.2 Information Retrieval 21

2.2.3 Knowledge Representation 22

2.2.4 Other Related Work 23

3 Introduction to Haystack **25**

3.1 The Data Model 25

3.1.1 The Data Graph and Straws 25

3.1.2 Straw Typing 27

3.1.3 Straw Subtypes: Needles, Ties and Bales 28

3.2 Services 30

3.2.1 Data Manipulation Services 31

3.2.2 Information Processing Services 33
3.2.3 System Services 34
3.3 Other Implementation Features 34

4 Discussion of the Data Model **37**
4.1 Goals of the Data Model 38
4.2 Representing Documents 39
4.2.1 What Is a Document? 39
4.2.2 Why a Document Is Represented by a Bale? 41
4.3 Collections 44
4.3.1 Containment vs. Reference 45
4.4 How Bales Express Relationships 47

5 Data Manipulation Services **51**
5.1 Event-Driven Services and Dispatching 52
5.1.1 Overview 52
5.1.2 Star Graph 52
5.1.3 How Dispatching is Done 54
5.1.4 Running Triggered Services 54
5.2 The Archiving Process 55
5.2.1 Archiving Service 56
5.2.2 Type Guessing 57
5.3 Data Processing Services 59
5.3.1 Directory Extractor 60
5.3.2 Tar Extractor 61
5.4 Implementation Details 62
5.4.1 Archive and Create Bale Methods 62

6 Profiling **67**
6.1 Theory Behind Profiling 67
6.2 Representing Profiles 69

6.2.1 The Profile Class 70
6.2.2 The WordIDMap Class 71
6.3 Computing Profiles 71
6.4 Profiling Service and Utilities 72
6.4.1 Storing Profiles 74
6.4.2 Viewing Profiles 74

7 System Design **75**
7.1 Promises and Haystack File 75
7.1.1 Promises 76
7.1.2 Haystack File 77
7.1.3 Promise Cache 78
7.2 System Services 80
7.2.1 Hayloft Management Service 80
7.2.2 Resource Control Service 82
7.2.3 Object Creator Service 83
7.3 Implementation of Straw Typing 85

8 Tasks for the Future **87**
8.1 Future Developments 87
8.1.1 Applications of Profiling 87
8.1.2 User Interface Improvements 88
8.2 Fixing Existing Implementation Problems 89
8.2.1 Robust Shutdown 89
8.2.2 Dependence on the ORO Package 89
8.2.3 Database Management 90

9 Conclusion **91**

A Data Model Implementation **93**
A.1 Needles 94
A.2 Ties 95

A.3 MIMEData Types 96

List of Figures

3-1 Data Graph Example 26
3-2 Data Graph Example with Complete Straw Types 27

4-1 The Current Way of Representing Relationships 47
4-2 An Alternative Way of Representing Relationships 48

List of Tables

A.1 Needle Types 94
A.2 Tie Types 95
A.3 MIME Types 96

Chapter 1

Introduction

Today, anybody who has access to a networked computer will testify to the awesome amount of available information and the frustration and pain of managing it. Indeed, finding a needed document, be it an email message or a web page, is often a time-consuming, if not an impossible task. A common approach to this problem is the use of tools that make it possible to search for documents by using key words. However, this approach is inadequate because people tend to associate documents with meta information, such as the author or the date, which is ignored or used poorly by most search tools. In addition, existing document management tools are unable to adapt to a specific user and lack adequate user interfaces.

1.1 Haystack Project

The Haystack project is an attempt to create a personal document management system that would address all of the above issues. Haystack utilizes the techniques already available to computer scientists and also innovates in a number of ways. The Haystack system is built on top of an Information Retrieval (IR) engine that allows indexing and searching of textual information. In addition to indexing a user's documents, Haystack collects and structures metadata about these documents. The discovery of an information structure creates an opportunity for novel approaches to presenting information. Finally, the ability to access a user's documents and to

monitor ways in which these documents are accessed, enables the creation of an intelligent system that can adapt to the user.

1.2 Implementation

Haystack project was launched two years ago, and its development proceeded in several stages. The first version of Haystack was implemented in Perl. Although the initial results were encouraging, Perl proved to be vastly inadequate to the needs of a large data-oriented system. Consequently, a second version has been implemented using Java. The new Haystack system has a powerful design that takes full advantage of Java's object oriented paradigm.

Haystack Design

One of the strengths of Haystack lies in the data structure used to store information about documents. This structure consists of a directed graph, in which nodes can represent both data and relationships between the data. Haystack defines its own notion of a Haystack Document as an aggregation of a document body and metadata. In the Haystack data graph, the Haystack Document and its body are represented by different nodes. Once a document becomes part of Haystack, a number of services take on the task of extracting all useful information from it and fully integrating the document into Haystack's data graph.

1.3 Goals of This Project

At the time when the author joined the Haystack project, substantial parts of the Haystack design and implementation were completed. However, a number of design issues remained unresolved and the implementation was not sufficiently robust. Both of these problems prevented further development of Haystack. The goal of this project was to address the existing problems and to extend Haystack's capabilities.

The initial stage of the Haystack design, described by Adar and Asdoorian [1, 2]

laid down the foundation of the Haystack data structure. This framework, also called the *data model*, sometimes lacked the specificity required to enable the cooperation of multiple services in the system. Many fundamental issues behind the data model were left unanswered. There were both a practical need and a theoretical interest to further explore the principles of the data model. In the course of the past few months, we deepened our understanding of the data model and made the necessary changes in the code to reflect this new understanding. This thesis describes these efforts and their results. It also discusses the reasoning behind the current data model.

Another major challenge of the project was to enhance the system robustness and to add new services. Since Haystack needs to manage a large amount of data, the system requires sophisticated software structures. Some of these structures existed and some needed to be created. A significant amount of work has been done to improve the Haystack system. This thesis describes this work and the parts that were added or significantly modified.

Finally, until recently, Haystack lacked the technology to support adaptation to users. As a means to tackle this task, we implemented the capability to create word-frequency profiles of a single document, a group of documents, or an arbitrary collection of text data. A word-frequency profile is a summary of a document's contents, which can be easily manipulated, compared to other profiles, etc. This thesis provides a description of the service and the structures that were created to support word-frequency profiling.

1.4 About This Thesis

Audience

This thesis is primarily intended for two groups of readers. First, it is intended for the computer science researchers and professionals who want to learn about the Haystack project and its results. Second, it is designed for the present and future developers of Haystack. The latter necessitates the inclusion of technical details not normally found in scientific publications.

The Structure of This Document

In order to accommodate as wide an audience as possible, this thesis provides some background on information retrieval and metadata storage. Chapter 2 provides this information and puts the project into the context of work done by other researchers. Chapter 3 introduces the Haystack system, its data model and services. Chapter 4 discusses several conceptual issues having to do with the representation of documents in the Haystack data model. Chapter 5 reviews the implementation of data manipulation services in Haystack, followed by Chapter 6 on the implementation of word-frequency profiling. Chapter 7 describes a number of internal data structures and services that are needed to enhance Haystack's robustness or improve performance. Chapter 8 outlines some of the tasks that lie ahead of Haystack developers in the near future. Chapter 9 summarizes and concludes this thesis.

Other Contributors

The work described in this thesis is the result of a collaborative effort of many members of the Haystack group, in particular, the leader of the group, Professor David Karger, and the two students responsible for the original design of Haystack, Eytan Adar and Mark Asdoorian. This thesis builds upon the constructs developed by these people. In addition, many insights about the data model came as a result of discussions that involved the entire Haystack group. Finally, parts of the implementation work were a result of collaborative efforts of Damon Mosk-Aoyama and the author.

Chapter 2

Background and Related Work

The Haystack project borrows its techniques from a number of computer science fields, some of which require brief introduction before we can proceed to describe the Haystack project. Section 2.1.1 introduces information retrieval, followed by Section 2.1.2 that discusses the notion of metadata and how it is used in Haystack. Finally, Section 2.2 reviews related work in the fields of IR, metadata representation and knowledge representation.

2.1 Background

2.1.1 Information Retrieval

Information retrieval (IR) refers to retrieving documents or texts with information content relevant to a user's information needs. Information retrieval includes two related, but different activities: indexing and searching. *Indexing* refers to the way documents and requests are represented for retrieval purposes. *Searching* refers to the way the files are examined and the items relevant to a search query are extracted. The two activities of indexing and searching have formed the focus of most of the research that has been carried out by the IR community. However, there is now increasing interest in complementary studies of the ways that people use IR systems and how user-system interactions should be organized to facilitate effective retrieval. While

indexing and searching are central to automated retrieval, they can support other forms of retrieval, such as browsing, which can also be enhanced by sophisticated visual presentation.

The Haystack project goes beyond basic indexing and retrieval and focuses on developing advanced data representation and visualization techniques. Haystack uses an off-the-shelf search engine, called ISearch [7], to implement basic indexing and retrieval. We sometimes refer to this search engine as the "underlying IR system." Haystack does not rely on any features specific to ISearch — so that this search engine can be easily substituted by another.

2.1.2 Metadata and Its Use in Haystack

A key feature of a good document system, such as Haystack, is the ability to represent information about documents and to express inter-document structure. Information about documents is called *metadata*. For example, a very important piece of metadata is the data format in which a document is stored (e.g. Latex, MS Word, plain text). Other examples of metadata are the document author(s) and the creation date. A user's annotation to a document is also considered a piece of metadata. This section discusses briefly the use of metadata in Haystack and then reviews related work on metadata representation.

Metadata can come from a variety of sources. It can be passed along with a document when the document is archived. It can also be extracted from the document (e.g. a title of an HTML page can be identified by an appropriate HTML tag). Finally, metadata could be generated by the owner of the document, as in the case of the user attaching a note with comments.

There are at least two ways in which Haystack uses metadata. First, metadata is used to search for a document. The IR system indexes metadata along with the text of documents. When a user queries Haystack to find a document, the ability to search metadata augments the user's ability to identify the desired document and to filter out undesirable ones.

The second way in which Haystack uses metadata is to establish relationships

among the user's documents. The fact that two documents have the same author implies a relationship between the two documents, and the Haystack data model makes it possible to capture this relationship. Similarly, if two web pages were visited one after another, it might be indicative of a connection between the two and this information should be recorded. Relations among the documents can also be considered metadata. The presence of relationships among the documents may be helpful to many tasks, including browsing the data graph to find a desired document.

2.2 Related Work

2.2.1 Metadata Representation

There have been a number of projects that address the use and representation of metadata. Most of these projects look at metadata from the point of view of information filtering or document transmission over the web.

One of these projects, the Resource Description Framework (RDF) [6], is an effort of the World Wide Web Consortium (W3C) to create an infrastructure to enable encoding, exchange and reuse of structured metadata. RDF is an extension of XML that creates a syntax that can be used to express metadata about the documents on the Web. RDF does not define any metadata — instead, it allows interested parties to create metadata schemas and define metadata semantics.

One of the applications of RDF is the Dublin Core Metadata Element [17] which defined fifteen standard fields to be used to describe a generic on-line document (e.g. title, author, language). The primary objective of the Dublin Core project is to facilitate discovery of electronic documents on the Web.

There are several key differences between how metadata is handled by RDF/Dublin Core and the Haystack data model. First of all, Haystack does not impose any schemas on its data model. In other words, Haystack does not limit the kinds of metadata that can be associated with a document. Haystack does not have to follow the formal approach for representing metadata taken by the RDF/Dublin Core because

Haystack is not concerned with the ability to share metadata with other users, and distribute it over the Web. The lack of such ability is, of course, a disadvantage.[1]The upside is that we can afford a flexible and informal data model — word searching does not require that semantics be defined and the human user, when browsing the data graph, is capable of extracting meaning from different syntactic constructs himself.

While most metadata models aim to allow authors to annotate documents with metadata for the purpose of document distribution, Haystack is not concerned with information transfer. Rather, Haystack's concern is to represent the metadata in a way that would make browsing the document space as convenient as possible. Hence, Haystack can afford a highly-flexible data model, without being concerned about standardization.

A number of metadata representation systems target specific, goal-oriented kinds of metadata. An example of such a system is PICS, which stands for the Platform for Internet Content Selection [5]. PICS is an Internet-based technology that gives Internet users control over the kinds of material to which they and their children have access. PICS makes it possible to label Internet documents. Labels can provide any kind of descriptive information about Internet documents, in particular they make it possible to rate a document according to its appropriateness for viewing by children.

PICS is an information filtering tool that enables users to find documents with the appropriate content. In this, the goals of PICS represent a subset of the goals of Haystack. In addition to improving search, in Haystack, metadata creates a linkage among the documents which provides a way for the user to find a document by browsing the document net.

Another system related to Haystack was developed at the Stanford University and is called Lore [4]. Lore is a database management system specifically designed for managing semi-structured information (i.e. structure is not schema-based). In this, Lore's data model is similar to that of Haystack.

Finally, there are a number of commercial document management systems that

[1]Note that given a set of metadata, Haystack could use RDF to define an appropriate schema and distribute it to other users.

allow a user to annotate and connect related documents (e.g., [8]). Usually, the role of metadata in these systems is marginal, and a system does not actively seek to expand the metadata set or use it to create inter-document structure.

2.2.2 Information Retrieval

Information Retrieval is relevant to Haystack in two ways. First, Haystack needs to enable the user to search documents and metadata. To this end, Haystack uses an external search engine (ISearch, [7]). Haystack does not attempt to improve the performance of an IR system per se. Instead, Haystack tries to make an IR system more useful to the user by enabling the user to search not only the document itself, but also metadata.

Traditional IR systems (e.g., [9, 14]) lack the ability to customize their behavior to a user. Haystack attempts to make up for that weakness by using IR techniques to implement user customization. This is the second way in which IR is relevant to Haystack. A number of academic projects attempted, with various degrees of success, to combine IR techniques with user customization. Stanford University researchers created a system [3] which learned about a user's preferences by having the user rate presented web pages. The system could then find more web pages of interest to the user. Another project in this direction was undertaken by a group at UC Irvine [11]. The Stanford and UC Irvine systems used the vector-based model, a standard for IR systems (e.g., [14]), to represent user interests. Haystack takes a similar approach to representing user interests (see Chapter 6), but is different in that it learns about the user from observing the documents in her possession and the documents she accesses on the Web. Note that the role of a Haystack user in the learning process is passive, unlike that of a user of the Stanford or UC Irvine systems, who is required to be active in order for a system to learn.

2.2.3 Knowledge Representation

Haystack can be considered a knowledge representation (KR) system. Unlike a general knowledge representation system, Haystack is primarily interested in knowledge about documents. Knowledge about non-documents, such as people, can also be represented in the Haystack data model, but the primary purpose of this knowledge is to serve as an intermediary in establishing relations among documents.

Although Haystack was not designed to be a pure knowledge representation system, classical texts on this field describe structures similar to those of the Haystack data model. For example, Reichgelt [13] describes a semantic net as a graph consisting of nodes and links. Links are unidirectional connections between nodes. Nodes correspond to objects, or classes of objects, in the world, whereas links correspond to relationships between these objects. This sounds very similar to what Haystack does, with nodes and links corresponding to Haystack needles and ties.

One of the major differences between the classical KR systems and Haystack is that these systems deal with data at much finer granularity than Haystack. A node in a KR usually represents a small piece of data, whereas in Haystack a large number of nodes are document bodies. A document, sometimes very large in size, is an aggregation of a large amount of unstructured data. We feel that "document" level granularity is appropriate for Haystack due to an end-to-end argument [15]. Documents are Haystack's input as well as its output (Haystack finds the document for the user, and the user finds needed information in the document herself). Thus, there is usually no need to artificially break up documents into smaller pieces for the sake of representation.[2]

[2]There are exceptions to this statement. It might be convenient to break a book into chapters because excessively large quantities of data are impractical in many respects, including that of information retrieval. In fact, the issue of the right degree of granularity at which data should be indexed, represented or returned to the user presents an interesting research topic.

2.2.4 Other Related Work

An interesting project, parts of which are related to Haystack, was conducted by Sheldon [16]. The project creates an architecture for information discovery based on a hierarchy of content routers that provide both browsing and search services to end users. The end user is presented with a document space that bears some similarities to that of Haystack. For example, documents in this system are organized in a tree structure. There is also an object representing a *collection* of related documents, which is something we considered (and rejected [3]) in the design of the Haystack data model.

[3]See Section 4.3 to find out why.)

Chapter 3

Introduction to Haystack

This chapter gives a high-level description of the Haystack implementation. The chapter consists of the following three parts. The first part introduces the Haystack *data model*, i.e. the structures used to represent documents and other knowledge. This part is followed by a review of services, i.e. functional components of Haystack. The last part familiarizes the reader with several features of the Haystack implementation that fall outside the scope of the first two parts but are still needed to understand the remaining chapters of this thesis.

3.1 The Data Model

3.1.1 The Data Graph and Straws

All information about documents is represented as a directed graph, sometimes referred to as the Haystack *data graph*. The nodes in this graph are called *straws*. Straws subdivide into three subtypes: needles, ties and bales. A *needle* represents a piece of "raw data." A needle is basically a wrapper around a Java object, such as a string or a number. A needle can also wrap around a file. A *tie* represents a directed relationship between two straws. For example, if straw A represents a document and straw B represents the person who wrote that document, an "Author" tie connecting A and B could be created to represent this relation. The reason that relations are

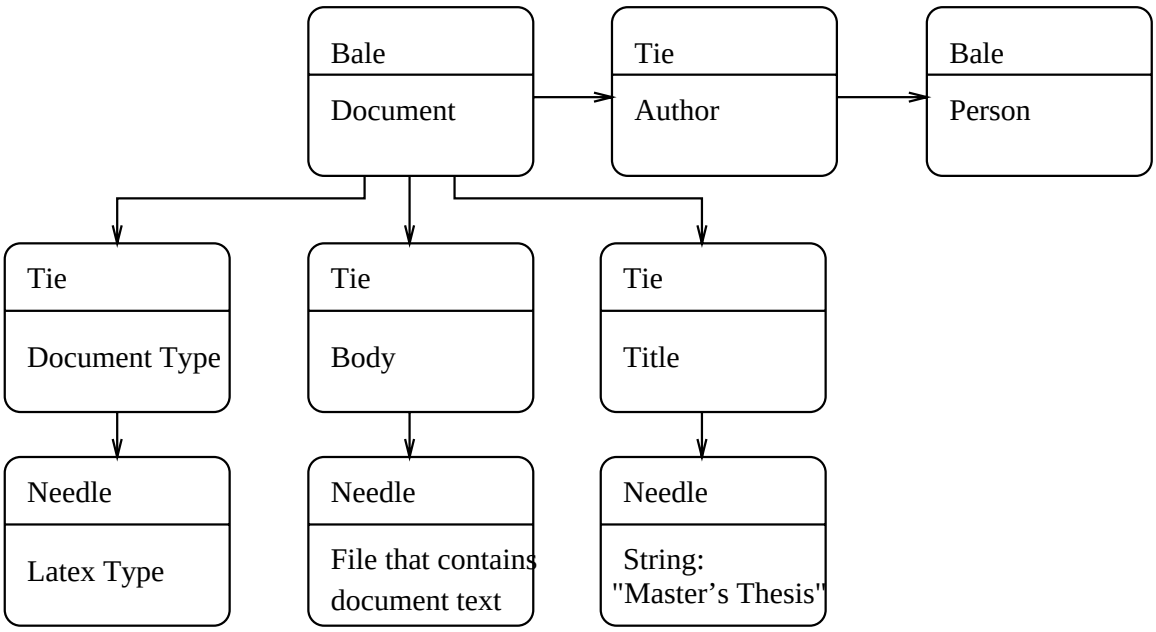

Figure 3-1: Data Graph Example

represented as independent nodes in our data graph is that we might want to point to or annotate a relation. For example, if a user creates a tie between two documents to indicate that they are related, he can attach to the tie a String needle with an explanation of why the tie was created. Finally, a *bale* represents a complex relationship among multiple straws. For example, a bale can be used to represent a document, a person or a query. These objects usually comprise several parts. For example a person object can combine the person's name, address and the date of birth. A bale is a centerpiece that connects these parts and represents their aggregation. Needles, ties and bales are basic constructs that could be used to represent any knowledge, including knowledge about documents. Note that the pointers that connect the nodes in the data graph do not carry any semantic information. Instead, if a relationship needs to be expressed between two straws, a tie is used to connect the straws and express the nature of the relationship.

Figure 3-1 shows an example of a data graph used to represent information about a document. In this example, the document is a Master's thesis stored in LaTeX format. Each box in the figure represents a straw. At the top of each box is the primary type

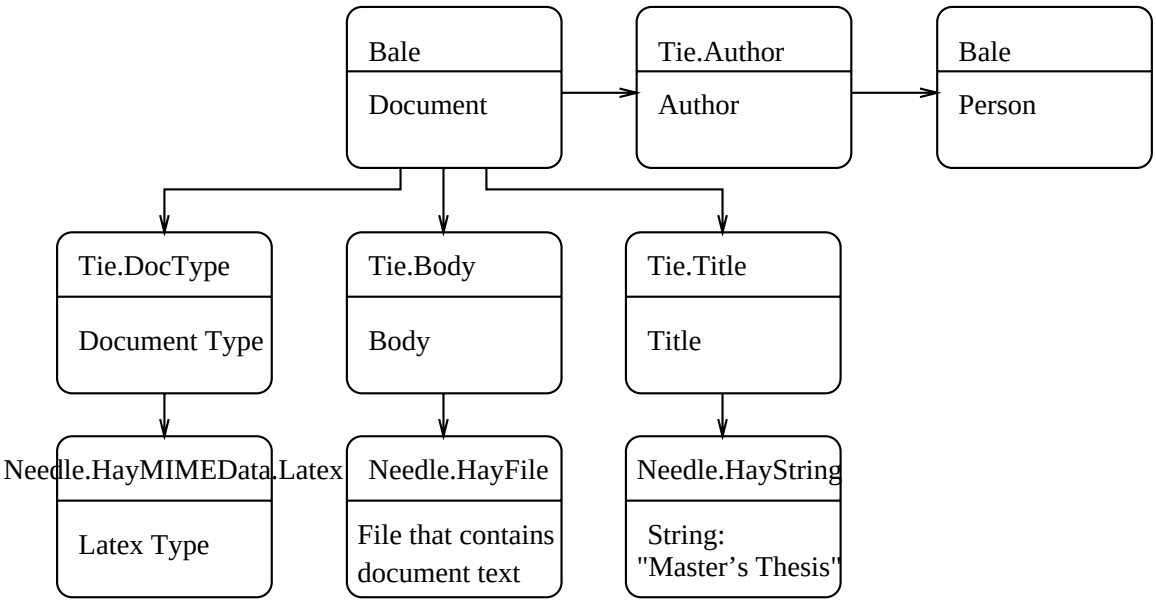

Figure 3-2: Data Graph Example with Complete Straw Types

of that straw, and at the bottom is a description of what the straw represents. Note that the bale representing the document and the needle representing the body of the document are different entities. The body refers to the actual text of the document. Besides the body, other information is present about the document, namely, its type (LaTeX), title ("Master's Thesis") and the author, represented by a person bale.

3.1.2 Straw Typing

Every node (straw) in the Haystack data graph must belong to one of the above-mentioned subtypes: a needle, tie or bale. These subtypes can further subdivide into sub-subtypes, so that straw types form a hierarchy similar to that of Java types. The type of a straw is expressed by a string, called a *label*. Examples of type labels are "Bale," "Tie.Author" and "Needle.HayFile.Text". The supertypes of a straw type can be derived from its label. For example, a straw labeled "Needle.HayFile.Text" is a subtype of "Needle.HayFile", which in turn is a subtype of "Needle." Each straw has a type label. The label is a final authority on the type of that straw. In the remainder of this thesis we will use the terms "label" and "type" interchangeably.

Figure 3-2 shows the same graph as in the previous example (3-1), except that the straw types are specified completely.

3.1.3 Straw Subtypes: Needles, Ties and Bales

Needles and Ties

In example 3-2 the needles represent "raw data" and the corresponding ties express the significance of the needles. To illustrate this distinction, consider the needle on the right containing the string "Master's Thesis." Its role in this graph is to express the title of the document. We could imagine a second document, a very short one, that consists of only two words: "Master's Thesis." The very same needle could serve as a body of that second document and be connected by a "Tie.Body" tie to the second document.

By having needles and ties, we can represent "raw data" and significance of data separately. Nothing about a needle should ever express the significance of the data in it. The needle label indicates the nature of the Java object encapsulated by the needle. The tie label indicates the type of relationship (significance) expressed by the tie.

In light of the above, I would like to explain a writing convention that might be confusing to the reader. We demonstrate it using the example in figure 3-2. In that example, we may sometimes write about the "Needle.HayFile" (middle at the bottom) as the "body needle." The word "body" in this phrase is used to identify a needle by its significance in the context of the document bale. Use of the term "body needle" should not be interpreted as an expression of some inherent property of that needle.

There is no predefined set of tie labels and a user can create ties with arbitrary labels. The only important thing about a tie label is that the party that creates a tie and the party that uses the tie have to agree on the meaning of the label.

The fact that the Haystack data model allows the user to create a data graph of an arbitrary structure sets the Haystack data model apart from the data model of a

relational database, in which a schema precludes a user from dynamically changing the structure used to connect the data (tables).

Immutability of Needles A needle encapsulates a piece of data (a Java object). Once a needle is created, the data may not be changed. The reason for this restriction, called *needle immutability*, is that Haystack might create other straws that point to the needle, or whose data is derived from the data in the needle. If we change the data in the needle, other straws or pointers might become incorrect. We choose not to change the data under the pointers.

Needle Uniqueness All needles in Haystack are unique, in that there are no two needles that encapsulate two identical Java objects. There are two reasons why needle uniqueness is desirable. First, Haystack strives to discover all possible relations among the elements of the data graph. If two needles have the same data, this implies a relation between the two. Of course, we could create a tie between them, but merging the two needles is even a simpler solution. For example, if two documents have the same title, the documents would be connected through the needle encapsulating the string with the title.

The second reason for needle uniqueness is the conservation of space. Although it might not matter for needles of small size, it would definitely be wasteful to store two large file needles with identical data.

Note that if for some reason Haystack did not merge two identical needles, that would not break Haystack or its data model. Needle uniqueness is a nice feature to have. Without it Haystack would still function, although its usefulness would be reduced.

Bales

We defer the general discussion of bales until the next chapter. However, there is one feature of bales representing documents that we must introduce now. If a bale represents a document, it must attach a "Tie.DocType" tie leading to a needle that

expresses the format in which the document body is encoded, e.g. Postscript. The DocType is an important piece of a document's metadata. The significance of the DocType will become apparent as this thesis progresses.

3.2 Services

The functional part of Haystack is implemented by *services*. A service is a functional Java class with a clearly defined set of duties. A service can either run at all times, or can be called in temporarily when needed. There are over a hundred services in Haystack that carry on a variety of duties, ranging from archiving and processing documents, to indexing and searching the data graph, to helping other services.

The services are bootstrapped by a module called the HaystackRootServer. This module is responsible for initializing, running and stopping services. We refer the reader to the Appendix A.2 of [1] for an in-depth discussion of the HaystackRootServer.

The goal of this section is to familiarize the reader with the services that will be relevant in the course of this thesis. It is not our goal to provide a comprehensive review of services on Haystack. Such a review can be found in [1]. The three main groups of services that we consider in this section are:

Data manipulation services archive documents, extract information from them, and put these data into the Haystack data graph.

Information processing services extract information from documents and create structures that could be used for user interface tasks. The services create word-frequency profiles, index files for searching, etc. Unlike the services in the previous category, information processing services do not modify the data graph.

System services assist other services in their duties, carry out a variety of system tasks (e.g. interacting with the database), manage the Haystack file repository, etc.

Other major groups of services in Haystack are the interface, observer and communication services. **Interface services** are responsible for the user interface. At the present time three user interfaces are available: web based, windows-based (implemented in Swing) and command-line. **Observer services** track the user's web browser, SMTP mail client and other gateways to actively archive user documents. **Communication services** are responsible for inter-Virtual Machine and and inter-Haystack communication. These parts of Haystack functionality fall outside the scope of this thesis.

Before we proceed to describe services in greater detail, it is worth mentioning that Haystack outsources some functionality to database management and IR systems. Haystack employs a database management module to persistently store its data. Haystack also uses an off-the-shelf IR system to index and search textual data.

3.2.1 Data Manipulation Services

Services in this group are responsible for creation and manipulation of the Haystack data graph. The following services comprise the group:

The Archive service coordinates the process in which a document becomes part of Haystack, e.g. creates a document bale and attaches needles with the metadata.

Fetch services fetch the body of a document given a location. For example, a URL fetch service obtains the file specified by a URL.

The Type Guesser service determines the type of a document (e.g. Postscript) when the document is archived.

Extractor services extract data from documents that contain or point to other documents. For example, a Gzip extractor uncompresses Gzip documents and makes the decompressed file a part of Haystack.

Field-Finder services extract information from archived documents. For example, a Latex field-finder extracts the title, the author and other metadata from a Latex document and attach appropriate needles to the document bale.

Textifier services extract text from formatted documents. For example, the HTML textifier gets rid of the formatting information in an HTML document (tags). The process of extracting text from a document is called *textification*. When a textifier service completes its work, the extracted text is placed into a needle which is attached to the document bale.

Similarity services identify similar documents. For example, the Similar Text service identifies documents with nearly identical text and connects the two document bales with an appropriate tie.

Many data manipulation services work by reacting to changes in the data graph. A service can indicate that it is interested in a change of a certain kind. When that change occurs, an appropriate event is generated. A dispatcher service, called Hs-Dispatcher, notifies interested data manipulation services of the event, which triggers the services to execute. For example, an HTML field-finder service wants to be called when a bale is created for an HTML document. At the time of initialization the HTML field-finder service specifies a structure, called a *star graph*, that expresses the pattern in the data graph that should trigger the service. This star graph is then passed to the dispatcher service. The dispatcher service keeps track of changes made to the data graph and, in the event that a pattern expressed by the star graph emerges, notifies the HTML field-finder of this event.

Data manipulation services may create chain reactions, in which addition of straws to the data graph by one service triggers other services to run. The chain reaction stops when all possible information is extracted and this information is put in the data graph.

Example: Data Manipulation Services in Action

The work of data manipulation services is best illustrated in action. Thus, we will describe a sequence of actions that Haystack takes upon a user's request to archive a document. Although this description does not include all of the data manipulation services, it should be of interest to the user in its own right because much of what

is described in this thesis was meant to improve the process below in one way or another.

In this example we describe what happens when a user issues a request to archive an HTML document at a specified URL.

1. The archiving service is given a URL location of a document; the Archiving service calls the URL fetch service to obtain the body of the HTML document.

2. The Type Guesser service is called to determine the type of the fetched document. Based on the ".html" extension in the URL, the type guesser determines the type to be "HTML."

3. The archiving services creates a bale to represent the document, and attaches metadata needles (location, document type) to the bale.

4. An HTML field-finder service reacts to the creation of a bale with document type "HTML". The field-finder extracts metadata from the body of the HTML document (title, author), puts the metadata into needles and attaches these needles to the document bale.

5. An HTML textifier service reacts to the creation of a bale with document type "HTML." The service removes HTML tags from the body of the document, puts the result into a needle and attaches the needle to the document bale using an appropriate tie.

3.2.2 Information Processing Services

Information processing services extract information from documents for specialized purposes, but do not make changes to the data graph. The information extracted by the services could be used for a variety of tasks, such as searching and customizing the user interface.

The Index Service indexes the text of documents and metadata with the underlying IR system. This operation makes it possible to subsequently query the IR system.

The HsProfiler Service creates word-frequency profiles of documents. It can also create a variety of custom profiles, e.g. a profile of all documents. The word-frequency profiles can by used for searching the documents and for other tasks.

3.2.3 System Services

System services help all other services carry out their duties. They also provide low-level management of the straw database. Below are some of the system services.

The Object Creator Service facilitates the creation of straws and ensures that various Haystack invariants are observed. For example, the service ensures needle uniqueness by verifying that no two identical needles are created.

The Persistent Object Service interacts with the underlying database to persistently store the data graph. The service also loads straws in memory when needed.

The Hayloft Management Service helps other services manage the directory that serves as Haystack's persistent data repository. This directory is called the *Hayloft*.

The Resource Control Service enables concurrency control. It provides other services with a mechanism to lock resources to prevent race conditions.

3.3 Other Implementation Features

Promises and HaystackFile

In theory, Haystack aims to manage a very large amount of data. Very often Haystack has to maintain several slightly different copies of the same piece of data (e.g., a body of a Gzipped (original) Postscript document, a decompressed version of the same document, and a version that has been textified). Instead of storing a piece of data, Haystack can use an object called a *promise*, that contains information on *how* to obtain that piece of data. For example, instead of storing an on-line document

internally, we can store a promise that contains the URL of that document. When the data in the promise is needed, the promise is *fulfilled*, and the data is returned.

In order to abstract a piece of data regardless of whether it is stored internally or can be obtained from a promise, Haystack implements a structure called *HaystackFile*. A HaystackFile might contain either a local file with the data, or a promise. The HaystackFile is transparent to its possessor in that the behavior is identical in both cases. Section 7.1 talks about promises and HaystackFile in detail.

Haystack IDs

In order to uniquely identify objects in Haystack, a service exists that can generate unique IDs, called *HaystackIDs*. Thus, all straws and promises have unique IDs.

Chapter 4

Discussion of the Data Model

Section 3.1 of the previous chapter gave an overview of the data model. This chapter provides an in-depth discussion of several important issues of the Haystack data model.

Haystack is a document management system, and the goal of its data model is to represent documents in a way that would enable the user to browse and search for the documents in the most efficient and intuitive manner. To this end, Haystack's data model makes it possible to express metadata and relations among documents, all of which create structure among the Haystack documents. Needles, Ties and Bales are the basic constructs that, in theory, allow us to build any knowledge system. In this chapter, we try to understand how the knowledge about documents can be expressed using these three types of Straws.

We begin by outlining some desired criteria for the Haystack data model in Section 4.1. We also give a sample list of objects that we might want to represent in Haystack. Since documents are the most important objects in our system, they receive a lot of attention in this chapter. Specifically, in Section 4.2 we discuss what a document is and why a document is represented by a bale. Then, in Section 4.3 we talk about another important class of objects — collections, and whether they deserve a special representation. There, we also discuss the difference between the relationships of containment and reference. Finally, in Section 4.4 we take another look at a bale as an encoding of a complex relationship.

4.1 Goals of the Data Model

Anything that is not a primitive type or a bilateral relation, is represented as a bale. Unlike needles and ties, bales are very general constructs and it is not obvious exactly how bales represent objects. To answer this question, we need to set out reasonable expectations of the kinds of objects that we might want to represent in Haystack. Below is a list of seven items that are representative of the objects we might want to represent in Haystack.

1. **Text document.**

2. **HTML page.** The key feature of an HTML document is that in addition to the text of its own, the document contains links to other documents.

3. **Book.** A large structured on-line document that can be physically represented in many ways — as a number of files with an index file referencing the files that contain the parts (chapters), as one large file, etc. An interesting feature of a book is that it represents a document, and at the same time its parts (chapters) are significant enough to be considered documents on their own.

4. **Tar archive.** Files of this type are created by a program called `tar`[1]. The purpose of this program is to group other files in one file. Normally, tar files are not expected to be viewed directly; archived documents must be extracted before they can be used.

5. **Directory.** A directory contains references to other files. It may be argued that a directory contains these files, but this depends on the definition of *containment*.

6. **Query.** This object represents the event of a user querying some knowledge base (e.g. web search query.) The two key parts of a query object are the query itself (presumably a string) and the set of objects returned as a result of the

[1] Tar stands for Tape Archive. For more information about `tar` see http://www.gnu.org/software/tar/tar.html

query. A query must be represented in the Haystack data model because a lot of useful information could be derived from it.

7. **Person.** Having a representation of a person is useful because people author documents. Also, a person is an example of an object that is not a document and it is important that the Haystack data model be flexible enough to represent objects that are not documents.[2]

4.2 Representing Documents

Documents are the most important objects represented in the Haystack data model. This section discusses documents and the way they should be represented. First, we need to define what we mean by the word "document". Once this is done, we explain why bales and not needles are chosen to represent documents.

4.2.1 What Is a Document?

The word "document" can be defined in many ways. Webster's dictionary defines a document as "a writing conveying information" or "a material substance having on it a representation of the thoughts of men by means of some conventional mark or symbol."

The Dublin Core workshop defined *document-like objects* (DLOs) by example. According to Weibel [17], an electronic version of a newspaper article or a dictionary is considered a DLO, whereas an unannoteted collection of slides is not. While acknowledging that DLOs might include all kinds of media (images, audio), Weibel says that "the intellectual content of a DLO is primarily text, and that the metadata required for describing DLOs will bear a strong resemblance to the metadata that describes traditional printed texts."

[2]Haystack does not attempt to mimic the real world. However, many relationships among the documents are derived from real world objects, and Haystack should be able to represent these objects as well, to the degree that this representation enables the expression of relationships among the documents.

We define a "real-world" document very broadly: a document is a representation or encoding of information. A document can originate in the real world — in a printed form, or in the electronic world — in the form of a file. In either case, before it can enter Haystack, the document must be represented as a computer file, called the *body* of the document.

This definition is far from rigorous — in fact, the term document is usually understood from examples, rather than from definitions. Of the examples on page 38, the first five items are considered documents — a plain text, an HTML page, a book, a tar archive and a directory. The reason that a directory is considered a document is that somewhere inside the operating system it is represented by a file (at least in many operating systems).

The key property of a document is that either a document body exists, or it is conceivable that such body could exist (directory). A person is not a document because a person could not be wholly represented by a computer file. Nor is a query considered a document, because a query, which we define as an aggregation of a query string and its results, is not a file. Note that under our definition of a document, it is moot for a document to have multiple bodies.

There are a lot of questions about the exact meaning of the word "document."

- Suppose that we have the same "document" saved as an MS Word and HTML files. Should these two files be considered to store the same document?

- Suppose that we have two copies of the same book. A reader wrote his comments on the pages of one book, the other has pages torn out. Should these two books be considered the same document?

There are no "right" answers to these questions and we will not try to answer them. The important questions for Haystack are: (1) how to represent a document and (2) once a representation is chosen, what exactly does it represent.

4.2.2 Why a Document Is Represented by a Bale?

Haystack Document

So far we have explored what we call a document in the "real world." In this subsection we write about the representation of documents in Haystack. What we represent in Haystack is somewhat different from the real world document — we call it a *Haystack Document* (HD). We define a Haystack Document as an **aggregation of the document body and the metadata**.

We must pick an object in the Haystack data model to represent a Haystack Document. The object representing a Haystack Document would attach straws representing the metadata. There are two candidates for the job — the needle that stores the document's body or a bale.

Why the Body Needle Cannot Represent the Haystack Document

In theory, a Haystack Document could be represented by a needle wrapping around the file containing the body of the document. After all, each real world document has a body, and is defined by this body. However, this needle cannot serve as such a representation for the following reasons:

- **Haystack does not need to possess a document's body to represent the document**. There are two situations in which we want to have a representation of a document without having the actual body.
  - We might know about a document whose body might be unavailable. For example, we might have a citation of a book, with its author, title, publisher, etc., yet not have the book itself. The fact that the body of the book is not currently available should not preclude us from representing the book.
  - It might be conceivable that a a body exists or could be created, yet Haystack does not possess it. For example, consider a directory. A directory could be represented as a file as it is the case in the Windows

operating system, or if the directory is fetched through a web server. At the same time, Java I/O does not allow the programmer to obtain the body of the directory — the programmer should use Java methods to query the operating system about the contents of a directory. To deny a directory a status of a document in this case would mean that a representation of a directory depends on the particulars of the computer system, which should be irrelevant to the question of whether a directory is a document or not.

Since a document body is not always present, and we want our representation to be consistent among the documents with and without a body, it is impossible to always represent a document as a needle.

- A needle could not possibly represent a document because it is nothing but "raw bits". A needle that contains a string "My Thesis" could be the body of one document and the title of another. Since needles are unique it would be extremely confusing if the needle represented a document and a title at the same time. In fact, metadata could not be attached to the needle directly, because the metadata is only relevant to the body in the context of the document.

Thus, the needle that contains a Haystack Document's body cannot represent the Haystack Document. Moreover, no other metadata needle can represent a Haystack Document for the similar reasons.

Having ruled out needles, a bale is left as the only plausible representation of a Haystack Document. In fact, a bale is a good representation because it was designed to represent a complex relationship and that is what Haystack Document is — it is an *aggregation* of the document body and the metadata.

Implications

The fact that a Haystack Document represents an aggregation of a body and metadata implies that there can be two different Haystack documents whose bodies are identical. Although not common, this is a plausible situation. It is conceivable that two people independently create two documents with the same text. To reflect the reality

accurately, Haystack *should* have different representations (two different document bales) for these two different real life objects.

Finally, we answer the two questions that were left unanswered at the end of the previous section, except that now we talk about a Haystack Document instead of a real world document.

- If we have the same "document" saved as an MS Word and HTML files, these would be archived separately, and represented with different Haystack Document bales. The two files correspond to two Haystack Documents because at least one of the pieces of their metadata is different — the DocType. Note that if the SimilarText service is running, it would see that the textified versions of the two documents are (nearly) identical and would create a "Tie.SimilarText" between the two bales.

- If there are two copies of the same book and a reader wrote his comments on the pages of one book, and some pages are torn out in the other, we would also have to create two different Haystack Document bales. Their bodies are slightly different (pages missing) and the second book has metadata (the reader's comments) that the first one does not. Again, most likely, a connection between the two books would have to be established eventually — by a SimilarText service, through a common Author needle, or in some other way.

We were able to answer these questions because semantically, a *Haystack Document* is closer to *a copy of a document* than to a *document*. Haystack can deal with this because storing "an extra copy" is cheap, and we hope the human user would figure out in which piece of information she is interested.

Immutability of Documents

Once a document bale is created, and a body needle is attached, this body may not be detached or replaced by another one. When a body needle is attached, Haystack services react to this event by creating additional straws based on the body of the document. For example, once a body of a Postscript file is created, an appropriate

textifier service creates and attaches to the bale a needle that contains the text of the body. If the original body disappears or changes[3], the new straws might no longer be correct. Since it is virtually impossible to determine which straws became incorrect, a body needle may not be detached or replaced.

In fact, the above argument applies not only to the body needle, but also to the DocType, location and other metadata needles. If any of these defining needles change, the Haystack Document would no longer be the same. The question then arises: what shall we do if one of the defining needles needs to change? Unfortunately, Haystack does not currently deal well with such mutations (but might be able to in the future).

A standard solution to the "mutation" problem is to create a completely different bale. By doing this, we introduce redundancy but avoid creating an incorrect data structure. Haystack relies on the user to ultimately decide which piece of data she wants to use.

For example. consider the following situation. Suppose a user archives a revision of a document, that is already represented in Haystack. Even if the user is not interested in keeping the body of the original revision, we would have to create a new document bale for the new revision. Of course, a tie should be created between the original document and the revision indicating the relationship between the two.

4.3 Collections

Another class of objects that could possibly warrant a special representation is that of collections. Among the examples given on page 38, a tar archive, a directory and an HTML document can be considered collections: a tar archive has associated with it a collection of files that it includes, a directory has associated with it a collection of files that it contains, and an HTML page has associated with it a collection of URLs to which it points.

[3]Due to needle immutability, the body could only change if the original body needle was replaced by another one.

While some of the examples listed above are commonly viewed as collections, HTML documents can be considered unusual. The reason for this is that a tar archive contains its parts, while an HTML document only points to other documents. We feel that while there is a distinction between containment and pointing (see Section 4.3.1 for a discussion of that), there are many commonalities — enough to say that URLs linked from an HTML page comprise a collection.

It is important to understand that under a broad definition collections are not limited to a few specific types of files. Any large amount of data must be organized to facilitate access. All data is grouped into collections (although sometimes implicitly) and might belong to more than one collection. Collections might be based on a variety principles. For example, data is grouped by the area of knowledge (chapters in the book), by ownership (all documents on a company web site) or by relevance to a specific person/object (links from personal home page).

Thus, a great variety of documents have the potential to be considered collections. What makes tar files and directories special is that the collection relationship has been the explicit purpose of the document. This argues that collection relationship among the parts should be stronger than in "implicit" collections. However, we feel that this difference has so far been insufficient to warrant an augmentation of the data model.

Thinking about collections led us to two other interesting questions. The first question is that of defining containment and reference. The second is that of whether one bale can express multiple relationships. We will now address these two question in turn.

4.3.1 Containment vs. Reference

The question of what constitutes containment and what constitutes referencing (or pointing) is interesting for two reasons. First, it comes up in the discussion of collections. And second, the answer is needed to label ties appropriately (if distinction is to be made in the Haystack data model).

Containment and pointing are not clearly defined with regard to many instances encountered in real life. For example, a file system directory is often viewed to contain

a file, unless it is a symbolic link, in which case, the relationship is considered pointing. Another example might be a book on-line. If a book is saved as a number of files representing chapters, and an index file points to the chapters, we say that the book points to its chapters. On the other hand, if an entire book is stored in a one large file, the file is said to contain the chapters.

The two examples above present at least two reasons to argue for no distinction between containment and pointing.

- The first example (directory) shows that containment and pointing are not clearly defined themselves with regard to many instances occurring in real life. While we can elaborate their corresponding definitions, this would require us to go into great technical detail.

- The second example (the book) draws our attention to the following problem. A single piece of knowledge (a book) can be viewed to have different relationships among its parts depending on the way it is stored in a computer system: a book can be stored (1) as one large file that *contains* the chapters, or (2) as an index file that *points to* the files that store the chapters. In both cases, conceptually, we are dealing with the same object — the book. However, it seems that the technical details of how the books are stored affect the way in which the book's internal structure is viewed (containment or pointing). We feel that a data model must represent knowledge unambiguously, which means that different representations of the same book must be conceptually identical. The only way to achieve this is to view containment as equivalent to reference.

Despite the reasons above not to make the distinction, Haystack continues to use "Contains" and "References" ties according to the following intuitive definition of these terms. A document A is said to contain document B, if the body of B could be obtained from processing the body of A (e.g., decompressing). Everything else is called referencing.

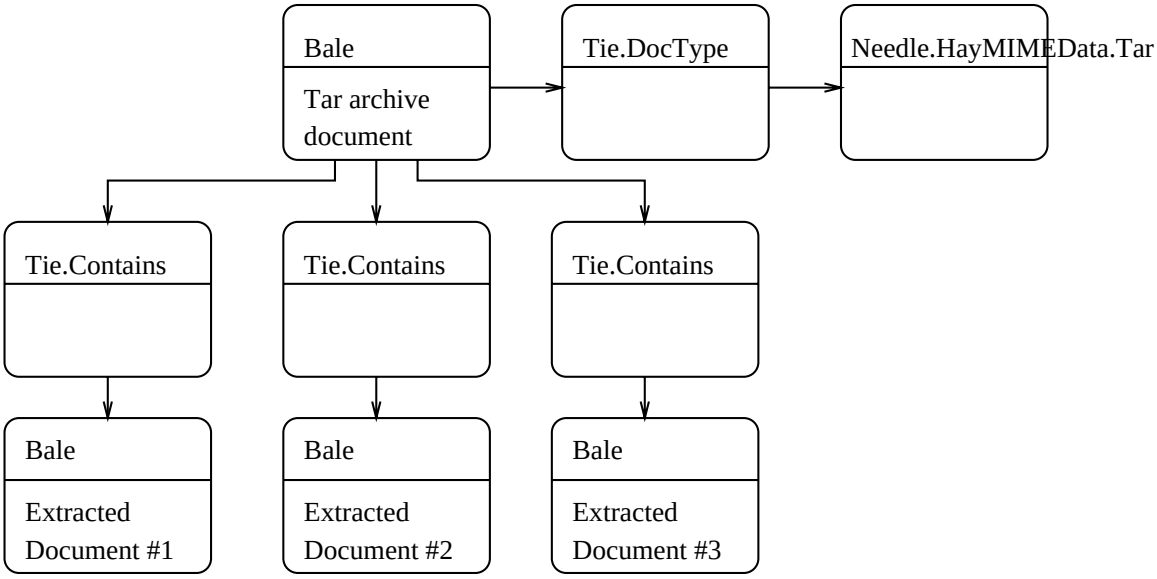

Figure 4-1: The Current Way of Representing Relationships

4.4 How Bales Express Relationships

The purpose of a Bale in the Haystack data model is to represent a multilateral relationship among straws. For example, consider a bale that represents an HTML document. In this example, we can talk about a relationship among all the documents linked from the HTML page. We can say that all of these documents are in the "referenced together" relationship. Similarly, for a tar archive, we could say that all documents extracted from the archive are united by a "common containment" relationship.

For an HTML or a tar document bale, the common reference or common containment relationship seems very significant. During the design of the data model we considered creating a separate bale that would link all of the documents pointed to or contained in one particular document. Figures 4-1 and 4-2 show possible representations of a Tar archive. Figure 4-1 shows the current representation. Figure 4-2 depicts an alternative representation where one bale represents the tar document, and another bale represents a collection of documents contained in the tar archive. We rejected this alternative representation for two main reasons.

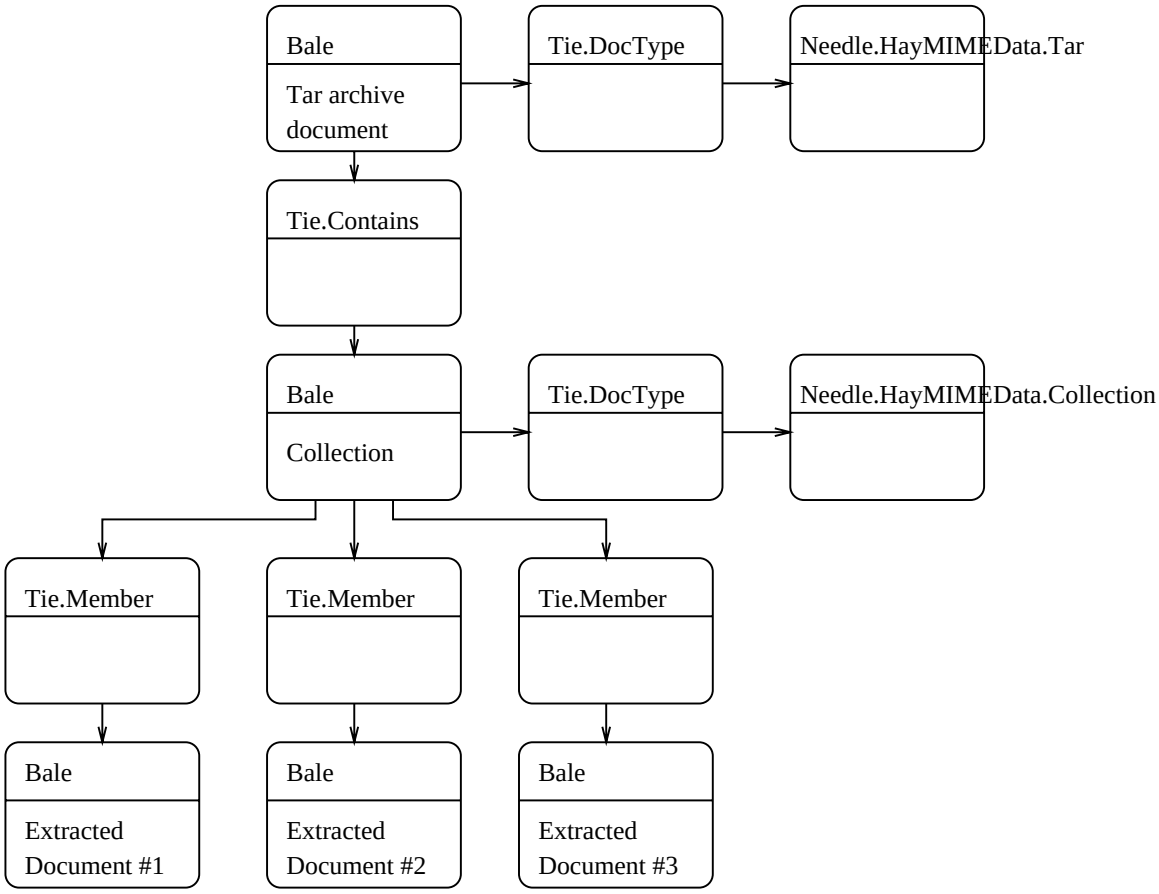

Figure 4-2: An Alternative Way of Representing Relationships

- The biggest reason for using a separate straw is the ability to point to it. For example, a tie requires a separate straw because sometimes we want to point to a relationship among two other straws, not at either of the straws themselves. In the example of a collection, the relationship of a collection is so closely tied to the document relationship that it is unlikely that someone would need to distinguish between the two.

- Unless somebody wants to point to the collection specifically, both representations are isomorphic in that one can be derived from another. For example, given representation 4-1, one can infer that all documents pointed to by the "Tie.Contains" tie are in the collection relationship. Since both representations are equivalent functionally, and representation 4-2 is more complex, we choose representation 4-1.

- Consider a paper that has multiple authors. We could say that these people define a "common authorship" collection. We could also talk about a collection of queries that match this document, etc. If we were to adopt representation 4-2, we would have to create bales for all "common ..." relationships or draw the line somewhere. The latter solution would make our data model explode in size without adding any real benefit. It would also mean special treatment for certain relationships which is undesirable.

The last argument illuminates the following point. If in a bale several ties of the same type are present, we can always talk about a sub-relationship among the straws pointed to by the ties of the same type. A bale can be said to represent one complex relationship, or to represent a number of simpler relationships. Although we could create a data model in which one bale would correspond to exactly one relationship, we refrain from doing so to avoid a complex data model and because under the current data model it is still possible to derive the simple relationships.

Chapter 5

Data Manipulation Services

This chapter describes the implementation of the data manipulation services and of the dispatcher that coordinates the work of these services. Data manipulation services are responsible for creating and modifying the Haystack data graph. The reader might remember from Chapter 3 that many data manipulation services are run in response to the events in the data model. We begin this chapter by describing the Haystack event model and its implementation in Section 5.1. After the mechanisms controlling the data manipulation services are understood, Section 5.2 describes the archiving service which coordinates the process in which a document becomes a part of Haystack. This process is called *the archiving process*. Next, Section 5.3 reviews the data manipulation services that are involved in the processing of information inside the documents. We take a closer look at the extractor services to give the reader a deeper understanding of how the data manipulation services interact with the data model. The last Section, 5.4, describes the implementation details of the archiving service.

5.1 Event-Driven Services and Dispatching

5.1.1 Overview

Haystack uses event dispatching to enable data manipulation services to react to changes in the Haystack data graph. Haystack implements its own event dispatching because Java event dispatching lacks flexibility required to support our needs. The events are dispatched by the HsDispatcherService. Once an event occurs, the dispatcher service identifies services that are interested in this event and puts them in the queue for running. The dispatcher has under its control several threads that are used to run the services in the queue. Services use a structure called the *star graph* to indicate their interests to the dispatcher.

Events

Haystack implements two main kinds of events — Haystack events and Object events.

- Haystack events occur when two straws are connected with a tie (HaystackCreateEvent), or when straws previously connected are disconnected (HaystackChangeEvent). HaystackChangeEvent is not presently used (i.e. events of this type are not thrown and no service is interested in this event).

- Object events occur when straws are created, deleted or changed. At the present time, none of the Object events are being used.

Thus, the only event presently used is HaystackCreateEvent. This event occurs when a tie is created to point from one straw, called the *source straw* to another, called the *target straw*.

5.1.2 Star Graph

The reason for having event-based services (as opposed to call-based) is the following. When a straw is added to the Haystack data structure, it is possible that further information can be obtained from that straw. It is also possible that the appearance

of a straw may signal that information may be extracted from another straw. For example, if a needle containing an HTML document is created, we know that certain information (e.g. the title) can be extracted from the body needle. In this example, the HTML field-finder service would be interested to know that such a needle has been added. There is a large number of services that might be interested in these types of events. Under these circumstances, an event-based model leads to a conceptually clear code model and easier application design. In particular, under the event-driven model, services are independent in that they do not need to know about each other.

A service needs to tell the dispatcher that the service wants to be triggered in response to a certain change in the Haystack data graph. This is done by specifying a star graph of interest. A star graph is an abstraction that describes a small piece of the data graph. A general form of the star graph is the following:

Straw Root of type R
    Tie of type T_1 connecting straw Root to straw of type S_1
    Tie of type T_2 connecting straw Root to straw of type S_2
    ...
    Tie of type T_n connecting straw Root to straw of type S_n

For example, consider the following star graph:

Straw Root of type "Bale"
    Tie of type "Tie.Body" connecting straw Root to straw of type "Needle.HayFile"
    Tie of type "Tie.DocType" connecting straw Root to straw of type
        "Needle.HayMIMEData.HTML"

An HTML field-finder service would use this star graph to indicate its interests to the dispatcher. When the field-finder passes the star graph in the example to the dispatcher, it says: "Inform me when the following configuration is created in the straw graph: a Bale that has a body of type "Needle.HayFile" and a document type of "Needle.HayMIMEData.HTML."

Note that each ray of a star graph consists of exactly one {tie, straw} pair. In other words, a ray can not be "tie X_1 connecting straw Y_1 connecting tie X_2 connecting straw Y_2" (two pairs). The reasons that a star graph does not have more than one

{tie, straw} pair are that (1) there was never a need for that, and (2) supporting rays with multiple pairs would significantly complicate the implementation.

5.1.3 How Dispatching is Done

When Haystack is initialized, all services inform the dispatcher about their interests by registering appropriate star graphs with the dispatcher. When the configuration of the data graph changes, the dispatcher determines whether the changed configuration matches any of the star graphs. Specifically, when the HsDispatcher service is informed that a HaystackCreateEvent has occurred, it determines whether the source straw and its out-coming ties fit any star graph.

For each star graph sg with root R, the dispatcher does the following. The dispatcher examines whether the source straw[1]has type R. For each tie-straw pair (T_i, S_i) in the star graph sg, the dispatcher examines whether the source straw has a tie of type T_i leading to a straw of type S_i. If all of these test results are positive, the service whose interest is expressed by the star graph sg is scheduled for execution.

Note, that when the field-finder is interested in Bales with bodies of type "Needle.HayFile", it is also interested in bales with bodies of type "Needle.HayFile.Text." The methods used to match a data graph with the star graphs need to be aware of this. These two methods are implemented inside of the StarGraph class.[2]The implementation of the star graph is fairly complicated, and the programmer needs to be extra careful when modifying that code.

5.1.4 Running Triggered Services

After the dispatcher determines which services are interested in the event, it places the {Service, Event} pairs into a queue for execution. The dispatcher has at its disposal

[1]Note that the dispatcher only looks at the star graph of the *source* straw and not at the star graph of the *target* straw.

[2]Until recently star graph matching had been done using the ORO package, which implemented regular expression matching. For a number of reasons we had to stop using ORO and matching had to be re-implemented using Java's String methods. The new implementation turned out to be simpler and also more efficient.

a pool of several threads which it uses to run the services scheduled for execution. In order to avoid synchronization problems, no two services working on two events with the same source straws should be executed at the same time. In other words, if services A and B are in the queue to handle events that share a source straw, the dispatcher must wait until service A is done, before running service B.[3]

5.2 The Archiving Process

The process of making an outside document a part of Haystack may be abstracted into two steps.

1. Obtain the body of the document (e.g., fetch a file from a URL location)
2. Create appropriate structures in the data graph to signify the document and the metadata. In particular,
   - Create the document bale.
   - Try to determine the document type (e.g. Postscript).
   - Attach all the metadata existing about the document at the moment (in the form of needles), including the document type, to the document bale.

The process outlined above is called the *archiving process*. Another way to define it is: steps that Haystack must take immediately after a request to make a document[4] a part of Haystack.

There are several possible scenarios in which an archiving process may be initiated:

- By a user, when he issues a request to the GUI to archive a file at a specified location.
- By a service. A service can learn about the existence of an outside document by analyzing a document already in Haystack (e.g., a directory that has already

[3]Note that when we say "running service x in response to the event y of type HaystackCreateEvent" we mean calling method handleHaystackCreateEvent of the class x with argument y.

[4]By "document" here, we mean a "real world" document as discussed in Section 4.2.1

been archived, and a service that decides to archive files in the directory). Alternatively, an observer service (see Section 3.2) can decide to archive a document (e.g., a web page browsed by the user).

Also, it is possible that a service already has the body of a document, but needs to perform step two to complete the archiving process.

5.2.1 Archiving Service

A service called HsArchive is responsible for coordinating the archiving process.[5] The HsArchive service is not a very large piece of code. However, this service is crucial to ensuring that the data graph is created in accordance with the rules of the Haystack data model.

In order to archive a document, a user must specify its location. At the present time there is only one kind of location — a URL location — other kinds of locations are not foreseen in the immediate future. Thus, from now on we will assume that location is represented by a URL. Note that a URL can specify either a remote document, or a local file (using the "file" protocol). Also, the reader should keep in mind that it is possible to archive a document that does not have a body (for example, a directory).

There are two main methods in the class HsArchive, called `archive` and `createBale`. These two methods implement steps one and two of the archiving process outlined at the beginning of this section. The `archive` method obtains the body of a document, and then calls the `createBale` method to create a bale and attach metadata straws to it. Note that the method `createBale` can be called by any service that needs to create a bale.

The implementation of the `archive` and `createBale` methods is relatively low-level, compared to the rest of this section. In order to preserve the high level of discourse, we defer the description of the implementation of these two methods until

[5] Other services can bypass HsArchive and create all the structures required for a new document. However, it is preferable that HsArchive be used for this task. Archiving a document can at times be tricky and it is better if this job is done by a "professional" service rather than an "amateur".

the end of this chapter (Section 5.4.1).

5.2.2 Type Guessing

As discussed briefly in Section 3.1.3, each document bale must attach a DocType needle. The DocType needle (e.g. the needle attached by the "Tie.DocType") indicates the format of the document represented by the bale. Examples of DocType needles are "Needle.HayMIMEData.Postscript" and "Needle.HayMIMEData.Directory". A special DocType "Needle.HayMIMEData.Unknown" is used to indicate that the bale represents a document whose type Haystack was not able to determine.

The DocType of an archived document is determined by a service called HsTypeGuesser. The "guesser" part in the name of the service indicates the degree of uncertainty associated with the process of determining the type of a document. This process is uncertain by nature because Haystack can not possibly know about all the computer formats. However, the type guesser attempts to determine some of the well-known formats as follows:

- If a URL location needle is present, the type guesser extracts the file part of the URL and, if the the file extension is present tries to determine the type based on that (e.g. ".ps" extension means Postscript).

- If a filename needle is present, the type guesser again looks at the type extension.

- When the body of the document is obtained through negotiations with a Web server, the server often informs the client about the nature of the data being transmitted (e.g. "HTML"). These data, called the *MIME content type*, is only present in files that have been obtained through the Web or by email. If this is the case with the document in question, the type guesser uses the content type to determine the DocType.

- Finally, the type guesser has the ability to determine the DocType by looking at the body of the document. However, such capabilities are not implemented at the present time.

The type guesser tries all four of the above methods[6], and assigns the appropriate DocType if at least one succeeds. If none of the methods succeed, the DocType "Unknown" is assigned to the bale.

The reason that we would rather have an "Unknown" DocType than no DocType at all is that some services might be able to extract data from a document of an "Unknown" type (e.g. a textifier service could use the "strings" command from the Unix operating system to extract text data from a file of an arbitrary type). If a DocType tie is missing, a service cannot determine which is true: (1) the type guesser has not run yet, or (2) the type guesser cannot determine the document type.

There are two alternatives as to how and when the type guesser can be run. First, it can be done the way it is implemented right now: `createBale` method calls the type guesser unless the DocType is already present. Second, the type guesser could be event-driven, i.e. the type guesser could trigger each time a new "piece of evidence" is added to the data graph: body, location, filename or the content type. We choose the first alternative for the following reasons:

- Since each document bale must have a DocType needle, calling the type guesser in the `createBale` method ensures that the rule is observed.

- It is possible that neither body, location, filetype nor content type ever become available. An event-driven type guesser cannot anticipate whether an "evidence" will be forthcoming, and will never be forced to assign an "Unknown" type.

- In practice, all four pieces of "evidence" exist by the time `createBale` is called. Thus, there is no reason to wait with calling the type guesser.

It has been argued that the user should be able to alter the DocType manually if he disagrees with the decision of the type guesser. This is problematic due to the issue of document immutability, discussed at the end of Section 4.2.2.

[6]If methods in the type guesser disagree on the answer, the type guesser picks the answer provided by the most "credible" method. The order of "credibility" is the following: based on body, based on content type, based on the filename, based on the location.

The last issue that needs clarification is what happens if we want to create a document bale, but we have neither the body of the document, nor its location, nor the filename, nor the content type. If this is the case, the caller of the `createBale` method can either pass the DocType among the straws to attach, or pass nothing, in which case DocType "Unknown" would be assigned to the document.

5.3 Data Processing Services

Data Processing services include field-finders, textifiers and extractors. Below is a list of existing data processing services.

Field-Finder Services extract information from archived documents, mostly metadata — the title, the author, etc. At the present time, field-finder services for the following document formats are available or being worked on:

- Latex
- HTML

Textifier Services extract text from formatted documents. At the present time, textifier services for the following document formats are available or being worked on:

- ASCII (dummy textifier)
- Dvips
- HTML
- Latex
- Postscript

Extractor Services extract data from documents that contain or point to other documents. *Extraction* here is defined broadly, and does not imply that extracted documents are contained in the parent document. Thus, although a directory *references*, rather than *contains* its files, the service that archives files in a directory is called an extractor. At the present time, extractor services for the following document formats are available or being worked on:

- BABYL (Emacs mail)
- Directory
- Gzip compressed document

- UU Encoded
- Tar archive

Although a significant number of changes were made to the code of all of these services, many of these changes were done to comply with modifications in other parts of Haystack. There were no significant changes in the way field-finder and textifiers work. Since the work of field-finders and textifiers was described well by Asdoorian and Adar [2, 1], it is not necessary here to repeat these descriptions. However, it would be beneficial to review the work of extractors for the following reasons. First, extractors were virtually non-existent at the time when Adar and Asdoorian wrote their theses. Second, many of the changes made to Haystack in the past half a year were inspired by the problems with the implementation of extractors. These problems stem mainly from the difficulties of interaction between the services and the data graph. By examining the extractor services here, we can illustrate the interaction between the services and the data graph, and prove that the current data and service models are, in fact, workable.

Thus, in the remainder of this section, we present two extractor services, Directory and Tar, which we believe to be representative of other extractors. Please note that in the description of the extractors, we talk about promises and HaystackFiles. If the reader feels that the introduction to these constructs in Section 3.3 was not sufficient, he is welcome to read section 7.1 that gives an in-depth explanation of promises and HaystackFile.

5.3.1 Directory Extractor

The directory extractor is an event-driven service that listens for the creation of a bale with a DocType tie leading to a needle labeled "Needle.HayMIMEData.Directory", and "Tie.Location" tie leading to a needle labeled "Needle.HayURL". When the star graph of interest is created, the method `handleHaystackCreateEvent` is called. The following is the sequence of actions taken by this method.

1. Obtain the URL of the directory from the needle that triggered the service.

2. Get a list of files in this directory using the Java I/O package.

3. For each file in the directory, do the following:

   (a) If the file is a symbolic link, do nothing.

   (b) If the file is not a symbolic link, call the `archive` method of the archiving service, passing the URL of the file as the only argument.

   (c) The `archive` method returns the HaystackID of the bale created for the new document. Using the Persistent Storage Service, get a pointer to the bale by its HaystackID.

   (d) Attach the bale of the new document to the bale representing the directory by the tie of type "Tie.References".

Note that this sequence of actions assumes the successful completion of each stage.

The reader should realize that the sequence above is only a framework of what the Directory extractor does. We can fine tune the service by creating rules of when the files in a directory should be extracted recursively, whether we might want to archive some files and not others, etc.

5.3.2 Tar Extractor

Tar archive extractor is an event-driven service that listens for the creation of a bale with a DocType tie leading to a needle labeled "Needle.HayMIMEData.Tar", and a body tie leading to a needle labeled "Needle.HayFile". When the star graph of interest is created, method `handleHaystackCreateEvent` is called. The following is the sequence of actions taken by this method.

1. Obtain the HaystackFile from the needle that triggered the service.

2. Run "tar -t" which outputs a list of files in the archive. Parse the output of the command to produce a vector of file names.

3. For each of the file names do the following:

(a) Create a promise that would extract the file with that file name from the HaystackFile containing the archive. Create a HaystackFile from that promise.

(b) Create two vectors, `archiveTieLabels` and `archiveStraws` that will hold the data to be passed to the archiving service.

(c) Create a needle with the HaystackFile that has just been created. Add the needle to the vector `archiveStraws`. Also, add the label "Tie.Filename" to the vector `archiveTieLabels`.

(d) Create a needle for the file name string. Add the needle to the vector `archiveStraws`. Also, add label "Tie.Body" to the vector `archiveTieLabels`.

(e) Call the `archive` method of the archiving service, passing to it two vectors, `archiveStraws` and `archiveTieLabels`. The `archive` method creates a bale for the extracted document, and returns the HaystackID of the bale.

(f) Using HsPersistentStorageService, get a pointer to the Bale by its HaystackID.

(g) Attach the bale of the new document to the bale representing the tar archive by the tie of type "Tie.Contains".

Again, this sequence of actions assumes the successful completion of each stage.

In step (a) we created a promise to extract a Tar file and then we fulfilled that promise in step (b). Note that the actual extraction is done in the promise.

5.4 Implementation Details

5.4.1 Archive and Create Bale Methods

This section complements Section 5.2 by describing the implementation of the `archive` and `createBale` methods.

Archive Method

The `archive` method has the following signature:

```
public HaystackID archive( URL loc,
                           Vector tieLabels,
                           Vector straws,
                           ArchiveOptions options,
                           HaystackUI theUI )
       throws ArchiveException
```

- `loc` is the URL location of the document to be archived
- `tieLabels` is the Vector of tie labels with which to attach straws
- `straws` is the Vector of straws to attach to the new document[7]
- `options` is the Archive Options object to direct the archiving process
- `theUI` is the User Interface (UI) to query for user input (a dummy UI is passed if the user is not involved in the archiving process).

This method is implemented using the following steps:

1. Make sure that neither body nor location are present in the `straws` vector.
2. Check whether the location `loc` has been archived before. If it has, use the GUI to ask the user whether to proceed.
3. Create a URL needle. Add it to the `straws` vector, and add "Tie.Location" to the `tieLabels` vector.
4. If the URL has a "file" protocol (e.g., "file:///projects/thesis/almostdone"), see if the specified location is a directory.
    - If it is, we know the document type and we know that body is not needed. Thus, we create "Needle.HayMIMEData.Directory" needle and add it to the `straws` vector, also adding "Tie.DocType" to the `tieLabels` vector.
    - If the specified location is a regular file, create a URL fetch promise for that location and create a HaystackFile from that promise. Put the HaystackFile into a "Needle.HayFile" needle. Add the needle to the `straws` vector and add "Tie.Body" to the `tieLabels` vector.

5. If the URL has a protocol other than "file", e.g. "http", create a URL fetch promise and create a HaystackFile from that promise. Put the HaystackFile into a "Needle.HayFile" needle and add the needle to the `straws` vector, also adding "Tie.Body" to the `tieLabels` vector.

6. Call the `createBale` method passing vectors `straws` and `tieLabels` to it.

Auxiliary `archive` method. Often, there are no straws to be attached to the new bale and the default archiving options and UI are to be used. As a matter of convenience, another `archive` method exists overloading the "main" `archive` method. The auxiliary `archive` method takes only one argument — the location. It then calls the "main" methods, passing empty vectors in place of `straws` and `tieLabels`, and default ArchiveOptions and UI in place of the `options` and `theUI` arguments.

Create Bale Method

The method has the following signature:

```
public HaystackIDs archive( Vector tieLabels,
                            Vector straws,
       throws ArchiveException
```

- `tieLabels` is the Vector of tie labels with which to attach straws
- `straws` is the Vector of straws to attach to the new document

Note that the elements of `straws` and `tieLabels` vectors should be in direct correspondence, i.e. the i^{th} straw in the `straws` vector should be attached with the tie whose label is the i^{th} element of the `tieLabels` vector. The following is done by the `createBale` method:

1. Create a new bale.

[7] This might not be the cleanest way to pass {tie, straw} pairs to the method, but it is by far the simplest in terms of implementation.

2. Create a needle with the current time and attach it to the bale using a "Tie.CreationDate" tie.

3. Attach all straws from the `straws` vector to the bale using ties with the labels specified in the `tieLabels` vector.

4. If a "Tie.DocType" tie is not in the vector `tieLabels` (i.e. the DocType is unknown), run the type guesser and attach the result with a "Tie.DocType" tie to the bale. If the type guesser fails to determine the type, the type "Unknown" is used. (see next Section 5.2.2).

5. The HaystackID of the bale is returned to the caller.

An ArchiveException is thrown if vectors `straws` and `tieLabels` have different sizes.

There are two advantages to a process in which first, all the needles to be attached to the bale are created, and then, all of them are attached. The more obvious advantage is that this adds to the conceptual simplicity of the code. The less obvious advantage is that we do not attach needles until we are sure that all other operations completed successfully, adding to the robustness of the process. If needles were attached one by one, and a failure occurred in the middle of the archiving process, we would be left with a structure that is partially complete yet missing some important components, which is undesirable.

What the second part of the previous passage was essentially saying is that we want bale creation to have the properties of a transaction. In other words, we either want a complete bale or nothing. In general having transaction capabilities could be very useful in Haystack. However, given that (1) nothing terrible would happen to Haystack if an operation, such as bale creation, is only partially complete, and (2) transactions are expensive and non-trivial to implement, it remains to be seen whether Haystack would ever employ transaction mechanisms.

Chapter 6

Profiling

Per-user customization has been a cornerstone of Haystack since inception. As part of these efforts, Haystack needs a means of expressing user interests and document relevance to these interests. These needs are addressed by using word-frequency profiles, a standard solution to these kinds of problems. A package of utilities has been implemented in Haystack to allow the computing and storing of word-frequency profiles. We begin this chapter by discussing the theory behind profiling in Section 6.1. The rest of the chapter describes the implementation of profiling in Haystack. The reader can refer to Section 8.1.1 in the chapter on future work for the examples of how profiles can be used.

6.1 Theory Behind Profiling

In order to evaluate the relevance of a document to user interests, we need to find a way to express these interests. Haystack makes an assumption that a user's personal files (and the documents she accesses on the Web) reflect on the interests of the user (e.g. a doctor will have many medicine-related files). Thus, a user's interests can be induced from the collection of her documents.

We need a way to express the *contents* of a document or a collection of documents to compare them to the contents of other documents. Information Retrieval relies heavily on the assumption that documents with similar contents use similar words,

i.e. same words will occur in both documents with a similar distribution of frequencies. To represent the contents of a document, we can compute its *countfile*, i.e. a histogram of words and their frequencies in the document. Similarly, by computing a countfile of a collection of documents, we can hope to express the contents of the entire collection.

The simplest way to measure the similarity between two countfiles is to compute their dot product:

$$\sum_{w_i \in W} f_1(w) f_2(w)$$

where W is the set of all words that occur in both documents, and $f_i(w)$ is the number of times word w occurred in document i. If this formula is used, countfiles with few common words would produce few non-zero summation terms, whereas countfiles with many common words would have many non-zero terms, and produce a larger result. Of course, there are many ways to improve the effectiveness of that formula, e.g. by normalizing. Since the details of the dot product formula may vary depending on the intended use, we will leave these details out of the discussion.

There are many ways to represent the contents of a document, the countfile being one of them. Depending on the intended use, a countfile may be modified (reduced in size, frequencies slightly changed) to improve its effectiveness for the task. We call such a modified version of the countfile a *word-frequency profile*, or simply a *profile*.

Haystack computes profiles which are different from countfiles due to the use of two standard IR techniques, called *stemming* and *stop word elimination*.

Stemming Stemming is the process of removing prefixes and suffixes from words. This is done to group words that have the same conceptual meaning, such as "walk," "walked," "walker," and "walking". Stemming maps such words to a common *stem*, which gives us two benefits:

1. If two documents use different forms of the same word, we would still be able to take that into account.
2. Stemming reduces the size of profiles, which makes storage and computation more efficient.

Haystack uses a freely available implementation of a *Porter stemmer*, which is a well-known algorithm for this task. Note that using stemmers is not without dangers. For example, a stemmer could mistake the last two letters of the word "numb*er*" for a suffix, and map "number" (incorrectly) to stem "numb", which would equate two semantically different words "number" and "numb."

Stopword Elimination A *stopword* is a word, such as a preposition or an article, that has little semantic content. A stopword can also refer to a word that has a high frequency across a collection. Since stopwords appear in many documents, and are thus not helpful for distinguishing documents, these terms are usually removed from the profile. This procedure is called *stop word elimination*. Some systems have a predetermined list of stopwords. However, stopwords can also depend on context. For example, the word "computer" would probably be a stopword in a collection of computer science journal articles, but not in a collection of articles from *Consumer Reports*.

One of the goals of my thesis project was to implement the basic capabilities needed for computing profiles of individual documents as well as profiles of groups of documents (or any collection of text for that matter). The rest of this chapter describes how this was implemented. Section 6.2 describes how profiles are represented in Haystack. Next, Section 6.3 discusses how profiles are computed. The last Section 6.4 reviews the profiling service that provides an interface for for dealing with profiles in Haystack.

6.2 Representing Profiles

A word-frequency profile is nothing more than a table that associates words with integer frequencies. However storing words as strings is inefficient. Alternatively, strings can be mapped to integer IDs and the profile can be a mapping of word IDs to frequencies. This solution is used in Haystack. The Profile class serves as a profile abstraction and the WordIDMap class is used for translation of words into IDs and

vice versa.

6.2.1 The Profile Class

In addition to the mapping of IDs to frequencies, each Profile contains a string, called the *info string*, that can serve as a description of the profile. The following is the main constructor for class Profile:

```
public Profile(int t[], int f[], String info)
```

- The array of integers `t` (t for *T*oken IDs) should contain the word IDs in increasing order. The ordering is used to make operations on profiles take linear time.

- The array of integers `f` should contain the frequencies corresponding to the word IDs in the array `t`.

- String `info` should contain a description of the profile.

The most important method of class Profile is called `addMultiply`. The method has the following signature:

```
public void addMultiply(Profile prof, int factor)
```

`addMultiply` adds profile `prof` multiplied by `factor` to `this` profile, where `this` is the profile on which the method is called. The ability to multiply by a factor allows for great flexibility in manipulating the profiles. For example one profile can be subtracted from another if the factor is negative. Note that a frequency can be negative to express the "irrelevance" of a word.

6.2.2 The WordIDMap Class

This class represents a one-to-one mapping of words to integers. It is used to assign integer IDs to words because using integers is much more efficient than using strings.

- The two main methods are `getID` and `getString`. The first method translates a word into an ID, while the second does the opposite. If an ID is requested for a word that is not in the map, a new ID is created transparently to the caller and that ID is returned.

- Once a mapping from a word to an ID has been created, the mapping can neither be altered not removed.

- Mapping in both directions takes constant time.

6.3 Computing Profiles

The ProfilingTools class is responsible for the actual computing of profiles. The three main methods of this class are:

```
public static Profile computeProfile(HaystackFile hfile, WordIDMap map)
throws IOException

public static Profile computeProfile(String str, WordIDMap map)
throws IOException

private static Profile computeProfile(Reader in, WordIDMap map)
throws IOException
```

These methods can compute profiles of a HaystackFile with text data, a String and a Reader stream with text data respectively. Each method accepts a WordIDMap, used to translate words into IDs. The map is updated if a method encounters a word

that is not in the map. Computing a profile takes $n \log(n)$ time, the bottleneck being the Quick Sort needed to put the Word IDs in order.

ProfilingTools are capable of eliminating stopwords and stemming. Stop-word elimination is done using a list of words obtained from [10]; the list is 319 words long. For stemming we use a freeware implementation of the conflation algorithm described by Porter [12].

6.4 Profiling Service and Utilities

Profile creation and management is controlled by the profiling service, implemented by the class HsProfilingService. The profiling service can compute two kinds of profiles — generic and needle.

Generic Profiles A generic profile is identified by a unique string key. We choose to use a string key instead of a numeric key (such as a HaystackID) because we want the key to be able to carry semantic information as well (in other words, we want to be able to name profiles). Profiling service can support an arbitrary number of profiles. A generic profile in Haystack can combine:

- Textified document bodies (stored in "Needle.HayFile.Text" needles).
- Arbitrary text data (stored either as a String or a HayFile needle).

Thus, the profiling service implements the following methods for dealing with generic profiles:

- `void addStringToProfile(String str, String key, int factor)`
  Adds String `str` to the profile specified by the `key`. The string's profile is multiplied by the factor `factor` before being added to the profile. If no profile is associated with the given key, a new profile is created that is associated with that key.
- `void addNeedleToProfile(HayFile needle, String key, int factor) throws LabelException`

Adds a text file needle to the profile specified by the key. The needle's profile is multiplied by the factor `factor` before being added to the generic profile. If no profile is associated with the given key, a new profile is created that is associated with the key. Note that the HaystackFile encapsulated by the needle is passed to the method, not the needle itself.

- `private Profile getProfile(String key)`
  Returns the profile associated with the given `key`. If no such profile exists, returns null.

- `public boolean discardProfile(String key)`
  Removes the profile associated with the given `key` from persistent storage.

Needle Profiles Creating a profile of a document is a more expensive operation than combining two profiles and it is preferable that a profile of a document be computed only once and then cached. To this end, there is a special kind of a profile, called the *needle profile* that contains a profile of a HayFile needle with text data (which takes care of all the large document bodies). Needle profiles are computed for needles of type "Needle.HayFile.Text". This profile is computed transparently to the user, by calling a `getNeedleProfile` method. When the method is issued for the first time, the profile is computed and returned to the caller. Needle profiles are permanently stored, so that on a subsequent call to `getNeedleProfile` Haystack does not need to recompute it. A needle profile is identified by the needle's Haystack ID.

The same map is used for all profiles. When the profiling service is initialized, it reads the WordIDMap from disk (if one exists). On shutdown, the profiling service checks if any new words have been added to the WordIDMap, and if so, it saves the modified WordIDMap to disk.[1]

[1] WordIDMap is always saved before a modified profile is saved. This is to make sure that a Haystack failure would not result in a profile on disk that uses a word ID that is not in the WordIDMap.

6.4.1 Storing Profiles

All profile related information is stored in the directory "$HAYLOFT/profiles", where "$HAYLOFT" is the location of the Hayloft, Haystack's permanent repository. The profile directory has two subdirectories, one for needle and the other for user profiles.

Initially we considered storing profiles of needles as a part of the Haystack data model, i.e. representing a profile as a straw of some kind. We rejected this idea for the following reason. A profile is not something a user will ever want to look at. Rather, it is a system object. Adding an unnecessary piece of data to the Haystack data graph would clutter information presentation and would make it harder for the user to navigate the graph.

6.4.2 Viewing Profiles

Two methods are available for viewing a Profile object inside Haystack. Method `printWordProfile` of the Profile class takes a single argument, a WordIDMap object, and prints the profile to the standard output. `printSortedWordProfile` does the same thing except that the words in the profile are sorted in the order of increasing frequency.

For debugging purposes profiles can be viewed from outside the Haystack as well. The `ViewProfile` command (`haystack.bin.ViewProfile`) takes two arguments, `word-id-map` and `profile`. The first argument specifies the name of the file containing the WordIDMap (presently, "$HAYLOFT/profiler/WordIDMap"), and the second the name of the file containing the profile. One can easily figure out the file name of the profile of interest by browsing the "$HAYLOFT/profiler" directory. ViewProfile outputs the profile with its elements sorted by frequency.

Chapter 7

System Design

This chapter reviews the last several pieces of system design that have been added or changed in the course of my project. Section 7.1 describes promises and the promise cache service. It also covers the HaystackFile construct that abstracts away a difference between a promise and a file. Section 7.2 talks about three important system services that deal with Hayloft management, concurrency control and straw object creation. This is followed by a discussion of the straw typing implementation in Section 7.3. The contents of this chapter are a bit scattered because we wanted to leave the implementation details until the end; we did not want to mix them with the more conceptual discussion in the previous chapters.

7.1 Promises and Haystack File

Computer users today have to deal with large amounts of data, which includes both their own files and the files available on-line. In order to manage all of these data, Haystack must store internally a large number of files. In fact, the amount of such data becomes prohibitive when one takes into account the following facts:

- Due to sophisticated data formats a large percentage of files have sizes of several megabytes.

- For each document, Haystack often needs to maintain several files (e.g. textified version) or might need to create additional documents (e.g. documents extracted from an archive).

- While the number of personal files is limited by the amount of persistent storage (i.e. hard-drive) available to the user, the number of files available on the Web is virtually infinite.

Often, files managed by Haystack are readily available (e.g on the Web) or Haystack knows how to obtain these files (e.g. Haystack can easily extract a file from an archive). Given this and the limitations on the amount of data Haystack can store internally, it becomes desirable be able to store not a file itself, but an object expressing the knowledge of how to obtain that file.

7.1.1 Promises

In order to provide this faculty, Haystack uses a structure called a *promise*. A promise is normally a small object that stores information on how to obtain a specific piece of data. When the data is needed, a method called `fulfill` is called on the promise, and the data is returned.

It is only useful to create promises for large pieces of data. Thus, we do not have a promise that returns a String or an integer.

A promise is implemented as an abstract class (src/haystack/object/Promise.java). Any class extending Promise must implement method `fulfill` that returns an `InputStream` with the data, or `null` if the promise could not be fulfilled. In addition, the Promise class provides a method that, given a file name, writes the data of the promise in that file.

Each promise has a HaystackID associated with it. The capability to uniquely identify a promise is used by the HsCacheService (see section 7.1.3).

There are many classes extending the Promise class. There are promises that fetch files from the web, promises that extract files from archives or compressed files, promises that textify, etc. Often, a service that is supposed to do something (e.g.

textify) only creates an appropriate promise, and it is the class implementing the promise that actually does the job (e.g. textification).

7.1.2 Haystack File

If Haystack has a piece of data, it can either store a file with this data, or a promise that returns the data when fulfilled. Note that promise fulfillment results in an `InputStream`, which, when serialized to disk, becomes a *file*. Thus, a promise can be viewed as a substitute for a file. It is desirable to have an object that looks like a regular file, yet may have a promise inside it.

Such an object exists and it is called a *HaystackFile*. A HaystackFile can either store a file or a promise inside it. Specifically, a HaystackFile can store a `java.io.File` object. As the reader may know, `java.io.File` is essentially a file name and does not have read or write methods. When the programmer wants to read the data from the `java.io.File`, she creates an `InputStream` from the File object, and the `InputStream` can then be read. Similarly, HaystackFile does not have a read method, but has method `getInputStream` that returns an `InputStream` with the data. Besides `java.io.InputStream`, the user can also request a `java.io.RandomAccessFile` using the `getRandomAccessFile` method.

Promises are read-only by nature. Since HaystackFile wraps around a real file or a promise, a HaystackFile is always read-only. HaystackFile returns only read-only objects: `InputStream` is read-only by definition, and `RandomAccessFile` is created with a read-only option.

Sometimes, there are many ways to obtain a piece of data. The same file can be stored in multiple locations on the Web, for example locations x and y. For the sake of robustness the user might want to create a HaystackFile that would first attempt to fetch the file from location x, and if that is not successful, attempt to fetch it from location y. In order to implement this, HaystackFile stores an array of promises. When the user requests data from the HaystackFile, Haystack attempts to fulfill all the promises in that array. As soon as one promise is fulfilled successfully, the data is returned.

A constructor for a HaystackFile takes either a `java.io.File` or a `haystack.object.Promise`. If the programmer has created a HaystackFile with a promise and wishes to add an additional promise to it, the `addPromise` method is used.[1]

HaystackFile Verification

It is not uncommon that a promise fails (e.g. when an object specified by a fetch promise is moved or deleted). When a promise is unfulfillable, it is preferable to determine that as soon as possible. When a promise is passed to the HaystackFile constructor, the program first verifies that a valid InputStream can be obtained from the promise. If promise fulfillment fails, the constructor throws an InvalidDataException[2]. Similarly, a `java.io.File` object may specify a file name of a non-existing file, but HaystackFile constructor verifies that this is not the case.

Verifying the input to the HaystackFile might have positive repercussions on the performance. A HaystackFile often becomes the data of a HayFile needle. If the data in the HaystackFile is invalid, creation of this needle triggers services that are certain to fail. This results in wasted system resources (time, risk of resources not being available to other services). Verifying the input to the HaystackFile prevents this waste.

A possible disadvantage of verification is that verification itself is an unnecessary waste of resources (we have to fulfill a promise). However, in reality the data of the HaystackFile is almost always requested shortly after the HaystackFile has been created and with the promise cache (described below) it is not such a big waste.

7.1.3 Promise Cache

It is very common that the data of the same HaystackFile is requested several times over a short period of time. If the HaystackFile contains a promise, this promise must

[1]Note that normally it does not make sense to add a promise to a HaystackFile that stores a `java.io.File` because presumably such a file can be accessed reliably

[2]Unfortunately, this verification does not protect from the case that a file that the promise depends on becomes unavailable in the future (e.g. web site with the file in question is shut down.)

be fulfilled each time the data is requested. Fulfilling a promise for a large file might be costly in terms of processor time, memory and I/O resources. In order to improve efficiency, the results of promises can be cached.

HaystackFile uses a service called HsCacheService to cache the data of the promises. Remember that each promise has a HaystackID and this ID can be used as a key for a promise in the cache. Promise caching works as follows. When the data is requested from the HaystackFile and the HaystackFile needs to fulfill a promise, it first checks whether the data of the promise is currently available from the cache. If this is the case, HaystackFile obtains these data from the cache and returns it to whomever requested it. If the data is not in the cache, HaystackFile fulfills the promise and passes the result to the cache service. Then HaystackFile returns the requested data.

There are two reasons why caching is done by a special service instead of HaystackFile itself. The first reason is that maintaining a cache is a complicated task (cache needs to be cleaned up every now and then, etc.) and it is preferable to have a special service responsible for it. The second reason is that the way the cache is designed, it can potentially be used by parties other than HaystackFile to store temporary files for a controlled period of time.

Implementation of HsCacheService

When a file is placed into the cache, it remains there as long as it is being used by somebody. If a file has not been used for a certain period of time (FILE_TIMEOUT), the file is removed from the cache. Note that the cache service does not have the ability to determine directly whether a file is still being used. Upon request for a file, HsCacheService creates an `InputStream` or `RandomAccessFile` object and then returns this object. The receiver of the `InputStream` may use it for a split second and then discard the object, or may keep it for the lifetime of the Haystack process. For a long time this technical difficulty precluded us from implementing a cache. Finally, a solution was found that uses `java.io.FileDescriptor`.

A FileDescriptor is an opaque handle to the underlying machine-specific structure representing an open file or an open socket. A FileDescriptor has a method

that tells whether it is valid (file is open) or invalid (file has been closed). Both `FileInputStream` and `RandomAccessFile` have FileDescriptors inside them. When the HsCacheService creates an `InputStream` or a `RandomAccessFile`, it retains the FileDescriptor of that object. After an `InputStream` object has been used, Java closes the stream, and the corresponding FileDescriptor becomes invalid. Each time the HsCacheService does a cleanup, it checks all FileDescriptors in its possession for validity. When all FileDescriptors for a particular file become invalid, the cache service knows that the file is no longer in use.[3]

Java specifications do not provide an excessive amount of documentation for FileDescriptors. For this reason, it is recommended that FileDescriptors not be abused by the programmer (e.g. cloned or used to create other objects) lest the cache service might stop working properly. The only piece of code that needs to use the FileDescriptor is the HsCacheService. Other services should avoid using FileDescriptors of the objects that might have come from the promise cache.

7.2 System Services

7.2.1 Hayloft Management Service

Haystack uses a directory called the *Hayloft* to store its persistent data. Some services create subdirectories in Hayloft and manage their files there. Often, a service needs to create a temporary file or directory, or store a file permanently in Haystack. A service called the HsHayloftManagementService helps other services manage these tasks.

The Hayloft management service maintains a temporary directory in Hayloft that can be used to create temporary files and directories. The following methods provide these capabilities:

- `public String createTemporaryDir()`

[3]When an InputStreamReader is created from a FileInputStream, the `InputStreamReader` has the same FileDescriptor as the FileInputStream.

This method creates a new temporary subdirectory in Hayloft. One of the uses of this method is by the extractor services that need a temporary directory to which to extract files. The temporary directory disappears on Haystack's shutdown. The method returns the absolute pathname of the new directory, or null in case of failure.

- `public String getNewTemporaryFileName()`
  This method returns a file name that could be used to create a new temporary file. The filename is guaranteed to be unique for the current Haystack session. It is also guaranteed that a file with this file name does not currently exist. The file is destroyed at the end of Haystack session.

- `public RandomAccessFile getNewTemporaryRandomAccessFile()`
  This method creates a new file in the Haystack temporary directory. The file is created with read-and-write permissions. This file is destroyed on Haystack's shutdown. Returns the file handle to the new file, or null if creation fails.

There are two ways in which Haystack can "possess" a file. The first is by having a promise, i.e. knowing a way to get the file. Promises are normally used for files stored remotely, or files that can be obtained by applying some method to another file (e.g. extraction). The second way is to store a file permanently inside Hayloft, i.e. save the file internally. This process is called *shelving*. Hayloft management services provide two utilities for shelving files:

- `public HaystackFile shelve(InputStream in)`
  This method saves data in the InputStream permanently in Hayloft. The method returns a HaystackFile object for this file in case of success, or null in case of failure.

- `public HaystackFile shelve(Promise promise)`
  This method takes a promise and saves its data permanently into some file in Hayloft. The method returns a HaystackFile object for this file in case of

success, or null in case of failure. Shelving a promise guarantees that its data will always be accessible in the future.

Shelved files are assigned arbitrary unique names and stored in the ArchivedFiles directory inside the Hayloft. We might move to a more sophisticated directory structure in the future because (1) we might run into limits on the number of files in a directory, and (2) a directory with a large number of files takes a long time to access.

7.2.2 Resource Control Service

Haystack is a fairly large system, and there are many instances when a programmer has to worry about the possibility of race conditions. Besides, Haystack essentially implements its own database system where synchronization is always a concern. To deal with this problem, the programmer can use HsResourceControlService which provides locking mechanisms by objects.

HsResourceControlService provides a capability to obtain and release exclusive locks. A lock is identified either by a string, or by a Straw object[4]. The following methods provide capabilities to obtain/release a single lock:

```
public void lockResource(String resource)
public void unlockResource(String resource)

public void lockResource(Straw resource)
public void unlockResource(Straw resource)
```

Very often a service might need to obtain several locks. In order to avoid the possibility of a deadlock, the locks should always be obtained and released in the same order. In order to facilitate this, several methods are available in HsResourceControlService. Although these methods are nothing but syntactic sugar, it is highly recommended that they be used to make sure that locking occurs in the right order:

[4]Which is mapped uniquely onto a string.

```
public void lockResources(String n1, String n2)
public void unlockResources(String n1, String n2)

public void lockResources(String n1, String n2, String n3)
public void unlockResources(String n1, String n2, String n3)

public void lockResources(Straw r1, Straw r2)
public void unlockResources(Straw r1, Straw r2)

public void lockResources(Straw r1, Straw r2, Straw r3)
public void unlockResources(Straw r1, Straw r2, Straw r3)
```

7.2.3 Object Creator Service

HsObjectCreatorService is responsible for the creation and permanent storage of Straw objects. Due to the existence of various kinds of Straws, as well as the need to enforce several data invariants, HsObjectCreatorService has grown to a significant degree of sophistication. This subsection will describe the changes made recently to support needle and straw label immutability.

Immutability Issues

Until recently, data in needles could be modified. After it was decided that data in needles should be immutable, we needed a way to enforce this. Specifically, the `setData` method had to be removed and needle data had to be specified at creation time. In order to accommodate this, creation methods of the HsObjectCreatorService had to be rewritten.

Similarly, when labels were introduced in all straws, and it was decided that labels should be immutable, creation methods had to be modified. In addition, we had to

ensure that the type of the data passed to a needle corresponded to one of the existing needle types. This condition could be checked either at run or compile time. For the sake of early error detection, we do the checking at compile time. To this end, separate methods were created, one for each type of Needle, that explicitly state the type of needle data in their signature.

Straw Creation Methods

As a result of all of the above considerations, the following methods had been implemented in the object creator service:

- `public Tie newTie(String label) throws LabelException`
  Creates a new tie with the specified label. Throws `LabelException` if the label is not a legal tie label (an example of an illegal tie label is "Bale").

- `public Bale newBale(String label) throws LabelException`
  Creates a new bale with the specified label. Throws `LabelException` if the label is not a legal bale label.

- ```
  public Needle newNeedle(String label, byte data[]) throws LabelException
  public Needle newNeedle(String label, Date data) throws LabelException
  public Needle newNeedle(String label, HaystackFile data) throws LabelException
  public Needle newNeedle(String label, Float data) throws LabelException
  public Needle newNeedle(String label, MIMEData data) throws LabelException
  public Needle newNeedle(String label, String data) throws LabelException
  public Needle newNeedle(String label, URL data) throws LabelException
  ```

  These methods create a new Needle of the specified type (e.g. the last method creates the object of type `object.needle.HayURL`). Explicit typing of the needle data (instead of passing an `Object` to a generic `newNeedle` method) prevents the programmer from passing an object for which a needle type does not exist. If needles were created with a generic `newNeedle` method, a run-time error could be possible. With explicit typing, error checking is done at compile time.
  ```

7.3 Implementation of Straw Typing

This section describes the implementation of straw typing in Haystack.

Java-based Typing vs. Label-based Typing

While the need for straw typing has never been disputed, various opinions existed on how to implement it. Two approaches could be used to implement straw typing. First, we could use Java typing. In other words, we could create a separate class for each sub-type of straws. Alternatively, we could implement our own typing. Label-bases typing is one possible way of doing that.

Until recently, Haystack had been using a hybrid of these two options. Although mostly Java typing was used, labels existed for some subclasses of straws. Haystack needs to support dynamic type creation (for example the user should be able to create a new type of a tie), and that is impossible with Java-based typing. However, Java-based typing was used whenever possible for the following two reasons: (1) label-based typing was believed to be significantly inferior to Java-based typing in terms of performance, and (2) Java already implemented inheritance and other nice features of typing.

As Haystack grew in size, the number of classes that implemented a particular straw type grew significantly. It was increasingly inconvenient to go through dozens of nearly identical files in order to implement a minimal change. In addition, compiling took a long time. Finally, we had to support label-based typing anyway for cases when labels where used. The near-consensus that emerged eventually was that Haystack was to stop relying on Java typing, and all typing had to be carried out through labels. This decision had a number of consequences. First, a large number of classes that essentially played the role of type placeholders were no longer necessary. Second, each straw object needed to have a label. Third, a number of services that used the properties of typing had to be revised, most notably HsDispatcherService. All of these changes were made in the past few months.

Right now all typing is based on labels. The move away from Java-based typing did

not cause a noticeable degradation of performance. Even though we still implement certain straw types as different Java classes[5] (for need of different functionality), this has nothing to do with the implementation of straw typing.

Immutability of Labels

Once a straw of a particular type has been created, the type of the straw may not change. This is because too many services make decisions based on the typing of that straw, and it would be hard to ensure the persistence of invariants after type change. As a way of ensuring that labels could not be mutated, the type of a straw (i.e. the label) should be passed to the straw constructor (see section 7.2.3 for details about how straws are created). Finally, no method exists to alter a straw label.

Label Naming Conventions

In order to avoid discrepancies among the developers, the following conventions are used when creating a label. A label must be composed of one or more sub-labels, concatenated by a dot, for example "Needle.HayFile.Text". A sub label may be a simple word ("Text") or a concatenation of several words ("CreateDate").

Haystack is case-sensitive when it comes to straw labels. All words that compose a sub-label must start with a capital letter (e.g. "Createdate" should not be used). In case of an abbreviation, e.g. "HTML", special care must be taken to make sure that word capitalization is used consistently.

[5]These classes are `object/Tie.java`, `object/Bale.java`, `object/Needle.java`, `object/needle/HayByteArray.java`, `object/needle/HayDate.java`, `object/needle/HayFile.java`, `object/needle/HayFloat.java`, `object/needle/HayMIMEData.java`, `object/needle/HayString.java`, `object/needle/HayURL.java`.

Chapter 8

Tasks for the Future

This chapter outlines some of the tasks facing Haystack developers in the near future. The reader should note that the theses by Adar and Asdoorian [1, 2] also contain a number of excellent ideas, some of which remain unimplemented. This chapter starts with a section on future developments which discusses possible applications of profiling and the improvements related to the user interface. The second section of this chapter deals with fixing some of the existing implementation problems, those of robust shutdown, dependence on the ORO package, and persistent storage.

8.1 Future Developments

8.1.1 Applications of Profiling

Applying profiling techniques falls outside the scope of this project. However, it is my hope that profiling capabilities will be used for user adaptation in the future. Below are a few examples of how that can be done.

- By combining the profiles of documents contained in Haystack, we can generate a profile of user interests in general. In fact, a first approximation of such a profile is already implemented in Haystack. The profile has key "AllTextNeedles" and it is computed from all needles of type "Needle.HayFile.Text". Once we have a user profile, we can use the dot product to assess the relevance of a

new document to user interests.

- The ability to create and manipulate word profiles leads to far-reaching capabilities for creating a highly perceptive information environment. For example, by keeping a combined profile of documents viewed in the past fifteen minutes, Haystack can be aware of the current activity of the user, and utilize this knowledge to sort results of Haystack or web search queries.

- Another use of profiles would be to help the user keep track of her activities on different days. Haystack can compute a profile of documents accessed on a certain day, and then select a few words most significant to that profile to summarize user activity for that day. Ability to keep track of user activities can help the user manage her time better.

- Haystack can apply user information expressed by profiles to improve Internet search and the search of documents in Haystack. We can use user information to post-process queries or pre-process query results.

8.1.2 User Interface Improvements

I believe that in the near future, the user interface (UI) will become a greater driving force in the development of core services. In order to streamline Haystack development, the core services should be modified early so that the capabilities needed by the UI will be ready as soon as they can be used. One of such capabilities is dynamic type creation. In particular, a user has to be able to create a new type of tie. Haystack maintains a list of legal tie types. Right now this list is static; it should be made dynamic.

Another possible improvement related to the UI has to do with Haystack's intention to improve the effectiveness of queries based on user's data files. To accomplish that, we need individual query logs. Then, we can use machine learning to analyze whether having user files can help improve querying. To this end, Haystack could implement, among other things, custom AltaVista and Yahoo query pages that would

remember the query and response, and also identify the person doing the query.

8.2 Fixing Existing Implementation Problems

8.2.1 Robust Shutdown

Since Haystack is a multi-threaded and event-driven application, graceful termination of all threads at user request imposes a number of requirements on the design of the system. All transient information should be kept in a format that allows for easy serialization and saving to disk. A central authority should keep track of all the running threads, and be able to signal to these threads that they must terminate. Since the threads may have to use resources managed by other threads, it is important that the latter threads be terminated only after the former complete their work. In order to ensure that all of these conditions hold true, Haystack programmers need to examine that (1) the services that bootstrap other services (HaystackRootServer, HsDispatcher, etc.) are implemented correctly, and (2) each service has a correct implementation of the `close()` method.

8.2.2 Dependence on the ORO Package

As was mentioned previously, Haystack has been relying on the package called ORO (Original Reusable Objects) to implement regular expression search. The creators of ORO seem to have abandoned their product, and it is unclear whether ORO will remain to work with the newer releases of JDK. I believe that Haystack should eliminate its dependence on ORO, and in the course of this thesis project, such dependence has been reduced significantly. One of the tasks for the near future is to amend the few remaining modules that still rely on ORO. ORO matching can either be replaced using the methods of class `java.lang.String`, or, if the matching is complex, by using one of GNU's regular expression packages.[1]

[1] Two regular expression packages are available under the GNU General Public License at `http://www.cacas.org/~wes/java/` and `http://www.crocodile.org/~sts/Rex/`.

8.2.3 Database Management

At the present time, straws are stored in a single file managed by the database management module, called DBM. This module has been borrowed from the freely available source code of W3C's Jigsaw web browser. I believe that the DBM module is inadequate to Haystack's needs. In particular, DBM's lack of scalability is a problem because Haystack aims to incorporate large amounts of data over long periods of time. Also, the DBM module appears to lack sufficient robustness, particularly at the time of Haystack shutdown, when all persistent data gets written to the disk.

It has been proposed to implement persistent storage of straws using a real database, for example, mySQL. I support this idea. However, I feel that the task may prove more complex than it appears to be and we should not attempt the task before a detailed design has been proposed and sufficient programming resources allocated. In particular, we must be careful not to repeat the mistake with ORO, i.e. getting involved with a product that faces the risk of being discontinued. We should also be judicious in using the database capabilities because having a data model of our own gives us the potential for innovation.

Another way we could use a database is to collect data on the usage of Haystack, e.g. query logs, browsing history, etc. Once Haystack begins to be extensively used, a substantial amount of such data will become available. Databases provide a standard solution for managing usage data. Subsequently, we could employ machine learning to improve Haystack's user adaptability.

Chapter 9

Conclusion

In the course of this thesis, we have described the structure and the implementation of the data model. We have also discussed a range of issues regarding the use of the data model. We established the role of a bale as a representation of a complex relationship that might also be viewed as a collection of smaller, homogeneous relationships. We also concluded that a document should be represented as a bale.

The move to label-based typing has reduced the number of files in Haystack by 20% and made it possible to dynamically add straw types. Label-based typing also simplified star graph matching, which is now 100% Java independent of any outside packages (ORO).

Haystack was made more robust by reorganizing and simplifying the archiving process. The extractor services now work, partly due to further specifications in the data model (no more confusion between the roles of ties and needles). Due to minor improvements in the code throughout Haystack, operations on the data graph are easier. Finally more error conditions are caught at compile-time as opposed to run-time, which should make Haystack more robust.

Lastly, a capability to create and manipulate word-frequency profiles is now in place, paving the way to greater user adaptability. In fact, a profile of all text documents of a user is already being computed and can be used as an expression of user interests.

All of these changes open the road to the work on user adaptability, novel in-

formation presentation techniques and more effective information retrieval. It is my hope that the work described in this thesis will endure, and that the full potential of the Haystack project will be realized.

Appendix A

Data Model Implementation

This appendix provides some implementation details of the data model. These details are primarily of interest to Haystack developers and they were left out of the main body of this thesis for the sake of the general audience. This appendix consists of three sections dealing with needles, ties and MIMEData types.

Table A.1: Needle Types

Needle Type	The object it wraps around	Description
Needle.HayByteArray	byte[]	
Needle.HayDate	java.util.Date	
Needle.HayFile	haystack.object.HaystackFile	A file or a promise (see section 7.1)
Needle.HayFloat	java.lang.Float	A floating point number
Needle.HayString	java.lang.String	
Needle.HayURL	java.net.URL	
Needle.HayMIMEData	haystack.object.MIMEData	

A.1 Needles

Needles wrap around "real" data objects — strings, numbers, files, etc. In fact, needles can wrap around any type of a Java object. Needle types can be identified by the kind of object they wrap around. Table A.1 shows the types of needles that are currently present.[1]

Note that all needle types are prefixed with "Hay" in order to avoid name collision between the needle class and the class being wrapped around.

[1]Note that types in Table A.1 are the *only* types of objects that can be stored in a needle. In order for a needle to be able to wrap around more types, new classes would have to be created. Also changes would have to be made to the object creator service and possibly other services.

Table A.2: Tie Types

Tie Type	Can Point From	Can Point To
Author	Bale	Needle.HayString
Body	Bale	Needle
Contains	Bale	Bale
DocType	Bale	Needle.HayMIMEData
Filename	Bale	Needle.HayString
CreateDate	Bale	Needle.HayDate
LastIndexDate	Bale	Needle.HayDate
Location	Bale	Needle.HayURL
MatchesQuery	Bale	Bale (Query)
References	Bale	Bale or Needle.HayURL
QueryResultSet	Bale (Query)	Bale (QueryResultSet)
QueryScore	Tie.MatchesQuery	Needle.HayFloat
QueryString	Bale (Query)	Needle.HayString
SimilarText	Bale	Bale
Text	Bale	Needle.HayFile or Needle.HayString
Title	Bale	Needle.HayString

A.2 Ties

A tie connects one straw to another and expresses the relation of the "from" straw to the "to" straw. Table A.2 lists some of the important tie types currently in use. This table is not complete but can give the reader a general idea of what kind of ties exist right now and how they are used.

There are conventions about when and how certain types of ties must be used. These conventions are needed so that services know which tie to follow to get certain data. For example, it is agreed that if a document bale has a location, this bale will have a "Tie.Location" tie leading to the needle expressing the location.

In general, we can say which types of straws a particular kind of tie can connect. The second and third columns in Table A.2 show the types of "from" and "to" straws. For example, the table indicates that an Author tie always leads from a "Bale" to a "Needle.HayString" (or a subtype of "Needle.HayString").

Table A.3: MIME Types

DocType	Explanation
Needle.HayMIMEData.BABYL	Emacs mail archive
Needle.HayMIMEData.Directory	
Needle.HayMIMEData.Dvi	
Needle.HayMIMEData.GZIP	GNU compress utility
Needle.HayMIMEData.Gif	
Needle.HayMIMEData.HTML	
Needle.HayMIMEData.Latex	
Needle.HayMIMEData.Postscript	
Needle.HayMIMEData.Tar	
Needle.HayMIMEData.Text	Plain Text
Needle.HayMIMEData.UUE	UU Encoded archive
Needle.HayMIMEData.Unknown	A document whose type is not known

A.3 MIMEData Types

Table A.3 shows a list of HayMIMEData subtypes presently used in Haystack. MIMEData subtypes are used to express document formats.

Bibliography

[1] Eytan Adar. Hybrid-search and storage of semi-structured information. Master's project, Massachusetts Institute of Technology, Department of Computer Science and Electrical Engineering, May 1998.

[2] Mark Asdoorian. Data manipulation services in the haystack ir system. Master's project, Massachusetts Institute of Technology, Department of Computer Science and Electrical Engineering, May 1998.

[3] Marko Balabanovic, Yoav Shoham, and Yeogirl Yun. An adaptive agent for automated web browsing. *Journal of Visual Communication and Image Representation*, 6(4), December 1995.

[4] Dallan Quass et al. Lore: a lightweight object repository for semistructured data. In *Proceedings of theACM SIGMOD International Conference on Management of Data*, volume 25, 2 of *ACM SIGMOD Record*, pages 549–549, New York, June 1996. ACM Press.

[5] R. Iannella. Metadata repositories using PICS. *Lecture Notes in Computer Science*, 1324:87–??, 1997.

[6] Eric Miller. An introduction to the resource description framework. Technical Report may98-miller, D-Lib Magazine, May 15, 1998.

[7] ISearch, `http://www.isearch.com/`.

[8] Blueridge, `http://www.blueridge.com/`.

[9] Alta Vista Search Engine, `http://www.altavista.com/`.

[10] Stop words from the University of Glasgow web site, `http://www.dcs.gla.ac.uk/idom/ir_resources/linguistic_utils/stop_words`.

[11] Michael J. Pazzani, Jack Muramatsu, and Daniel Billsus. Syskill and Webert: Identifying interesting web sites. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 54–61, Menlo Park, August 1996. AAAI Press / MIT Press.

[12] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[13] Han Reichgelt. *Knowledge Representation: an AI prospective.* Ablex Publishing Corporation, 1990.

[14] G. Salton. *The SMART Retrieval System - Experiments in Automatic Document Processing.* Prentice Hall, 1971.

[15] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[16] Mark A. Sheldon. *Content Routing: A Scalable Architecture for Network-Based Information Discovery.* PhD thesis, Massachusetts Institute of Technology, 1995.

[17] Stuart Weibel. Metadata: the foundations of resource description. Technical Report july95-weibel, D-Lib Magazine, July 1995.