Flexible and Robust Data Storage and Retrieval in the Haystack System

by

Kenneth D. McCracken

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

Author .....
Department of Electrical Engineering and Computer Science
June 11, 2001

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

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

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

Flexible and Robust Data Storage and Retrieval in the Haystack System

by

Kenneth D. McCracken

Submitted to the Department of Electrical Engineering and Computer Science
on June 11, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis we create a new backend to the Haystack information retrieval project. We call this flexible and robust module the triple store.

Haystack brings the power of personalized search into the hands of the user. It currently uses another data storage backend, but will eventually use the triple store instead. Several problems were faced while designing the new module. Many were due to the inflexibility of existing Haystack modules. We have created a new Haystack Trust Model that makes some legacy Haystack code more flexible.

The interface to the triple store allows data and metadata to be saved as labelled, directed binary relations. We hope that the interface is flexible enough to eventually save the Haystack Data Model, and robust enough to eventually make Haystack more dependable. The JDBCStore implementation of the interface connects to an interchangeable third-party database. It manages the more complicated robustness capabilities provided by the third-party database rather than handling robustness from scratch. Once Haystack switches to the JDBCStore backend, we will have a more dependable IR tool on which to develop other interesting research.

Thesis Supervisor: David Karger
Title: Associate Professor

Thesis Supervisor: Lynn Andrea Stein
Title: Associate Professor

Acknowledgments

I would like to thank my thesis advisors, David Karger and Lynn Stein, for exposing me to the intriguing technologies related to the Haystack project. I would also like to thank my family, Dad, Mom, Scott and Ali, for helping me meet the challenges of MIT over the last several years. I would also like to acknowledge the people who brought me sanity and good times in the midst of all the pressure, including but certainly not limited to the East Cambridge gang and the MIT swim team. And thanks to Joel Rosenberg for taking the time to help me with my writing.

Contents

List of Figures

Chapter 1

Introduction

Haystack is an information retrieval engine designed to help users search their digital information space. This thesis creates a new data storage layer for Haystack that we call the triple store[1]. The triple store provides a uniform way for clients to store flexible information. We call the module in Haystack whose specific task it is to enhance a user's search capabilities the root server. The root server will become the principal client of the triple store.

One thing computers are good at is storing large amounts of data. However, conventional software does only a mediocre job of helping users search that data. As hardware improves while costs drop, the volume of digital information available to users increases dramatically. Unfortunately the more information there is the harder it is to search through it. Haystack provides a personalized tool that helps users navigate their corpus of information. It does so by collecting information about the user's context before a search is conducted.

The triple store provides a physical representation of data that survives shutdown so that its client may safely return to a useful state. This thesis provides a uniform way for clients to write down their state. At the same time it provides a way to protect this storage from its clients by restricting access to a small set of tasks.

[1] We use this style of text when referring to Haystack-specific modules and other design entities throughout the body of this paper.

1.1 The Problem

The goal of the Haystack project is to provide an information retrieval tool that puts the power of searching in the hands of the user. The power of a Haystack search is measured in terms of its ease of use and its ability to point out data relevant to the context of a query.

It should be relatively easy to write Haystack data in the triple store. A lot of Haystack data is metadata. Metadata is data about data. In Haystack it may take the form of an author of a particular document or a user's annotations on that document. Haystack metadata is often expressed as relations that connect data together with a labelled link.

The triple store was designed with Haystack, its primary client, in mind. A requirement of the triple store is that it provide the ability to transfer Haystack data into it without breaking any of the system. Complying with legacy code proved to be quite a challenge indeed.

The basic data structure in the triple store is an ordered triple. In terms of Haystack metadata, each triple contains a pointer to the source and sink together with the string label of a relation. The triple store also allows storage and retrieval of a set of bits, such as a document's body. The meaning of document text stored in Haystack is in its relationship to other data, or, using triple store terminology, the meaning is located in the triples about the bits. Metadata can be attached to the data stored in the triple store just as the author of a document can be attached to a document in Haystack.

Design and implementation of the triple store focuses on several goals. A triple store has been developed that

- adheres to many general Haystack design principles.
- can coexist with Haystack without breaking any part of it.
- maintains a web of trust that helps dynamically loadable internal processes communicate while protecting them from external harm.

- allows multiple copies of itself and other modules to run concurrently.
- provides a flexible storage layer for Haystack data.
- provides a robust interface on which a correctly designed Haystack can depend for information storage.
- provides a stepping stone for Haystacks to begin communicating with each other.

By achieving these goals the triple store has been added to the repertoire of useful modules in the Haystack project. A fortuitous side effect is that the triple store may be used by clients other than Haystack's root server. Another benevolent side effect is that there is now a way to introduce new modules in Haystack and to modify existing rigid code to become flexible and reusable.

The triple store implementation connects to a third-party database that is not part of the Haystack project. We do not want to require that Haystack users connect to any particular database in order to run Haystack. Haystack code is written primarily in the Java language [33]. The triple store uses a Java package called JDBC [25, 29] that allows standardized statements to be executed on a large number of different database backends, maintaining Haystack's ability to run on many platforms. We also want to minimize the overhead of setting up the triple store to use this database. After a user can log in to his or her database, he or she simply configures the triple store with a JDBC driver name and a pointer to the database.

Haystack contained a lot of useful functionality prior the creation of the triple store. In order to preserve this functionality, the triple store does not break legacy Haystack code. Rigidity of existing design made adding a new low layer very difficult in some cases.

Several small processes called services run within Haystack to help improve information storage and retrieval. We want these services to be dynamically loadable and unloadable as they are needed. As services evolve, we want to restrict their ability to damage other services and parts of the Haystack Data Model. We introduce a utility as a generalization of the service idea. Utilities have the same dynamic properties of

services but aren't part of the Haystack root server module. Utilities may be part of any module in Haystack. We introduce the Haystack Trust Model that utilities can use for communication with trusted utilities and for protection from untrusted utilities. This trust model is somewhat similar to the trust model used for services. Nothing explicitly prevents services from adopting the Haystack Trust Model in the future.

One negative side effect of the Haystack Service Model is the requirement for uniqueness of services running inside Haystack. The triple store uses some services that were once restricted by uniqueness, such as those in the Haystack communications module. These services performed tasks that are useful to modules other than the root server. We have relaxed their reliance on a root server and renamed them as utilities. If a utility registers in a module's trust area it is restricted by uniqueness within that trust area only. Since we permit creation of multiple trust areas, we can have multiple copies of utilities that work within different modules concurrently.

The root server will become a large-scale client of the triple store. The root server stores most of its information according to the Haystack Data Model. The triple store was designed to permit physical storage of that model. It is flexible enough to store the many different types of Haystack data and metadata. The triple store's flexibility comes in part from the uniform way all information is stored. A client can write down unprocessed data as well as processable metadata descriptions.

The root server will use the triple store to give its data a more permanent existence. We want Haystack to be able to remember all of the things a user has placed into his or her Haystack, as well as all it has learned about the user. In the event of an unexpected failure, we want Haystack to be able to return to a recent useful state. The triple store provides the ability for a multithreaded client like the root server to protect its internal data. To do this the triple store supports transactions that can be used to separate one series of actions from another. Just like we need to protect services from damaging each other, we also need to protect threads from damaging physical data shared with other threads. The triple store provides transactions that a multithreaded client can use to maintain a meaningful state during execution, shutdown, and failure. It remains the responsibility of the client to use these transactions correctly.

The triple store's uniform methods will help make its data more portable. Although personal Haystacks currently can not collaborate with each other, we expect to add this feature in the future. To use the triple store, Haystacks must express all meaningful data using the uniform data structures of triples and unprocessed bits. It seems natural that in the future two Haystacks could transfer knowledge using versions of the same descriptive methods they used to write their triples and bits in the first place.

1.2 Overview

This thesis discusses how the design goals of the previous section were achieved. We try to identify weaknesses of our solution whenever possible. We also present the alternatives considered where appropriate. We hope that the changes made through this thesis will help Haystack become a more dependable, and therefore more useful, search tool.

The next chapter describes some of the current research technologies that Haystack and, more specifically, the triple store use. We then describe in Chapter 3 the details of the Haystack project with a focus on how the root server uses the Haystack Data Model and the Haystack Service Model to represent and manipulate its data. The Haystack Data Model is important in understanding the kinds of data the triple store should store. The Haystack Service Model made implementing the triple store difficult. One of the requirements of a triple store is to allow multiple instances of its server to coexist with the root server. We may want to use these triple store instances for future content versioning support. It was impossible to allow multiple concurrent triple stores and to reuse Haystack code written under the Haystack Service Model at the same time. The Haystack Trust Model, a solution to this problem, supports dynamic loading of multiple instances of modules, as presented in Chapter 4. The most important improvement is that Haystack code developed according to this new model can actually be reused elsewhere in the system, unlike Haystack service code. The triple store adopts this model to establish a web of trust for its utilities. The

`TripleStore`[2] interface is described in detail in Chapter 5. This interface defines the ways that clients can use the triple store. Transactions are very important to proper use of the triple store. A transaction contract describing the way in which transactions should be used is included with the interface. We next describe how the `JDBCStore` implementation of Chapter 6 performs the `TripleStore` methods. The `JDBCStore` knows how to connect to a database using a JDBC driver and how to translate the `TripleStore` methods to this driver. The implementation of the transaction contract is very important in ensuring that the triple store behaves correctly. Chapter 7 presents some relevant future research directions that could improve the triple store and the rest of the Haystack project. We conclude with some of the lessons learned from this thesis. Some lessons involve dealing with the difficulty of taking an idea and designing it to work with an existing system. We also have an improved understanding of Haystack.

[2]We use this style of text when referring to actual Haystack code throughout the body of this paper. Since Haystack code is Java code, we try to follow Java conventions in naming packages, interfaces, classes, variables, and methods. Packages use only lowercase letters in their names. Usually interfaces and classes have only the first letter of every word in their name capitalized. Variables and methods tend to capitalize only the first letter of words following the first word.

Chapter 2

Background

Before we get into the specifics of the Haystack project, we first address some related technologies. Haystack attempts to improve the search technologies available today by running a variety of services which aid in information storage and retrieval. The new triple store layer is a place to store Haystack data consistently and persistently using transactions. This new module also permits storage of flexible data from a standard descriptive modelling framework. This chapter presents some of the background helpful in understanding Haystack and the triple store.

A certain level of familiarity with mathematics and computer engineering concepts is expected of the reader. Some technologies such as graph theory, set theory, client-server architecture, and object-oriented programming design are out of the scope of this thesis. The author has attempted to avoid requiring knowledge specific to the Java programming language wherever possible. However, there are places that refer to Java because design decisions made use of Java-specific capabilities. If the reader is unfamiliar with Java it may be beneficial to consult a tutorial [33] or a more complete reference guide [9].

2.1 The State of Search

As the amount of available digital information grows, the problem of searching for relevant and useful resources becomes more and more difficult. Haystack provides a

personalized way for users to organize the corpus of information available to them. As computer users turn more and more towards remote information repositories, the already difficult task of manually organizing one's personal information space becomes unmanageable. All these systems become more useful if we can locate the right ones and retrieve and understand useful data from them.

Searching information was relatively painless when the Internet was comprised of individual personal computers [20]. These computers usually had a somewhat less permanent connection to the network. Communication often involved contacting machines point-to-point and following the links each provided. Files could be retrieved once a host was found via transfer protocols such as ftp [18].

We would like a tool that can search our own machines and the network to expose the information available. Common search tools don't achieve this goal because they do not understand their users [20]. They do not understand natural language questions. They can't correct mispellings. Users are often forced to live with the same canned results that everyone gets for a particular search. Search engines offer little customization. The Haystack project uses an alternative approach to searching. It will offer context aware searching of local Haystacks, communities of Haystacks, and arbitrary resources available on the network. Haystack pays special attention to the dynamic and often transitory nature of information resources.

This section develops the current state of search. We anticipate the addition of new services that enhance Haystack by performing tasks similar to the contemporary research. We must protect the triple store from services currently in Haystack and those we anticipate in the future.

2.1.1 A Bird's Eye View of Information Retrieval

Information retrieval (IR) is most well developed in the domain of text retrieval. Text retrieval can be used to search through a large corpus of documents. Information retrieval tools for other types of information resources are less well explored. Several available text retrieval techniques can be applied to the task of retrieving other data types [19]. As more multimedia resources in different formats become

available, perhaps more search technology in these domains will emerge. Currently efforts focus on text or other metadata that can be used to describe resources for which we have no means of searching their actual content.

The IR community commonly measures the effectiveness of information retrieval in two dimensions called precision and recall [24]. Precision measures how relevant retrieved resources are to queries. Recall measures how many of the relevant resources available actually appear in the results of a query. Typically an information retrieval system has to settle for some balance between the two. It is difficult to retrieve only relevant resources without missing something important. It is also difficult to retrieve all relevant resources without also retrieving some irrelevant ones.

Active areas of Haystack research study different IR models that attempt to improve both precision and recall [34]. Whatever the ultimate IR solution, Haystack will want to precompute representations of documents for query result processing. The precomputed data structures will be stored in the triple store as metadata about each document.

2.1.2 Catering to the User

Any search engine is useless if it doesn't find relevant information for the user easily. Incremental steps have been taken towards developing a more useful search engine. Improvements focus on increasing the accuracy of searches while reducing the user overhead.

One stepping stone toward better search is including vertical search tools. A vertical search tool is specialized to fulfill a particular type of search. Suppose I want to find a stock quote. Many search engines contain a link to a special search form for financial users. The tool itself may execute a query across a different subset of databases. Many general search engines offer versions of vertical search tools [31].

The data-centric approach also offers some favorable improvements to search. Similar to vertical search, data-centric search has special databases constructed as authorities on a particular topic area. When a user types a query asking about shows playing at the Boston Symphony Orchestra, the entertainment database is contacted.

The argument in favor of the data-centric approach says that while anyone publishing something on the Web could use metadata, they won't. Metadata could be used to place the data in the right subject area automatically. One cannot assume that metadata will be written, as it adds a lot of user overhead to Internet publishing [35]. Weaknesses of the data-centric approach include their inflexible structure and the overhead of creating a new topic database.

The task-centered approach tries to shift search results towards what the user is trying to do. It proposes that instead of using naiive collection-oriented solutions that search tools would benefit from knowing the user's task [17]. If I am trying to research information about Haystack, the commonplace assumption that I am trying to buy something is not accurate. I want to research Haystack, not buy a bushel of hay.

The designs presented above all hint at trying to get more information about what the user actually wants in his or her search results with minimal overhead. Wouldn't it be great if a search tool existed that could infer things about a query based on the context in which the user is asking his or her question?

2.1.3 Context-Aware Search

Haystack takes search one step further. Services maintain contextual clues to improve search. One such service allows users to annotate documents with comments that will help them locate that document in the future. Future queries will search across the comments as well as the documents themselves. We anticipate the creation of new services to maintain more contextual clues. New services may infer personal context by either watching what a user has done in the past or what he or she is doing right now.

Suppose I frequently use my Haystack IR tool to ask questions about computers. If I then ask a query with the term "apple", Haystack can use my past searching context as a clue that I want information on a certain type of computer. What if I am currently writing a document on cooking? Further contextual clues may direct the search otherwise. Suppose I have a system like Watson [6], that derives context

from the documents I am editing in my word processor. It may provide search results that are specific to a type of fruit. Suppose I am navigating through a result set that combines a mix of computer and fruit resources, but am only interested in those about fruits. SearchPad [4] helps the user navigate by allowing users to refine search and mark relevant search results. Improvements to Haystack that incorporate similar context awareness could dramatically improve the user experience.

2.2 Using Transactions to Achieve Robustness

As an application that stores information, Haystack needs a robust physical representation. Once Haystack has adopted the triple store layer, the physical representation will be used in the event of shutdown or failure to rebuild Haystack's application state. We present a model in which Haystack can safely issue a series of changes and queries to the triple store inside a single coherent transaction. A transaction is "a collection of operations on the physical and abstract application state" [10].

2.2.1 The ACID Properties

Transaction management helps reduce the general problem of maintaining persistence and consistency to the problem of adhering to ACID transaction properties of Atomicity, Consistency, Isolation, and Durability. The triple store cannot ensure ACID-compliance in the Haystack application that will eventually use it. It can only provide an ACID-compliant interface to disk. The ACID properties will remain a concern as Haystack migrates towards reliance on the triple store.

Atomicity means that the sequence of actions that constitute a transaction appear as though either all or none of them occur. Consistency means that a transaction must perform a state transition such that if the original state was consistent with constraints before the transaction, the new state will also conform to those constraints [10, 28]. An application that begins in a consistent state will, by induction on transactions executed on its state, always be consistent. Isolation means that while parallel transactions may be executed concurrently, none of the effects of their composite na-

ture will ever be visible [28]. For any two transactions A and B[1], the effects are the same as if either all composite actions of A occurred entirely before B or vice versa. Durability means that transactions that complete successfully survive failures [10]. Persistence is a synonym for durability.

We say that a transaction commits if it has renounced its ability to abandon execution [28], and will eventually execute in its entirety. We say that a transaction aborts if it has decided that all changes it has made will be undone [28], so that it is as if the transaction never existed.

An ACID-compliant transaction, hereafter referred to simply as a transaction, must be able to handle failures at any point. The term commit point refers to the instruction step within a transaction where the decision to either commit or abort a transaction is made. Until the commit point has been reached, our transaction is a pending transaction. Any instant in the execution of a program may constitute a point of failure.

Consider the effects of a power outage on a PC. Once a failure has occurred, the only way to rebuild the state of an application is from physical storage, here, a hard disk. The application must have the capacity to recover from the failure using only that physical data. After proper recovery the application's abstract state will again be consistent.

Recovery usually involves undoing any transaction that has not reached its commit point. For the sake of argument let us assume that on recovery we care only about completing those transactions that have been committed, and would like to effectively abort all aborted and pending transactions. The committed transactions must be able to run through to completion. The aborted transactions and those transactions still in their pre-commit phase must be completely undone. For all points up to the commit point of a transaction, we must be able to undo all changes that have been made.

Once the decision has been made to commit, the transaction must be able to complete the actions required on all resources it effects. Transactions should request

[1]We use this style of text when referring to example and abstract methods and variables throughout the body of this paper.

```
depositpay:   procedure(account, amount);
              account = account + amount;
              return;

paybill:      procedure(account, amount);
              account = account − amount;
              return;
```

Figure 2-1: The *depositpay* and *paybill* procedures

the ability to interact with these resources during the pre-commit phase [28]. If we wait until post-commit, we may not be able to access a resource to perform an operation that we are required to complete.

There are several ACID issues with which we must contend if we want to build a low-level interface to disk. However, the triple store uses third-party software to bridge the gap between memory and disk. Instead we focus on maintaining persistence and consistency in the application layer. Some issues with which we are concerned include isolation and the transaction specifications of a Java DataBase Connectivity API known as JDBC [25].

2.2.2 Enforcing Isolation While Using Transactions

Haystack is a multithreaded application. Multithreaded applications have the potential for more than one transaction to run simultaneously. Recall that transactions require only that after each transaction has committed or aborted, the system is in a consistent state. If two threads want to perform operations independently on a particular variable or address, locking may be required. The address can be an address in memory or on disk, depending on the context of the variable. A lock is a mark made by one thread to protect the address from being read from or written to by another thread [28].

Consider the pseudocode in Figure 2-1 that defines the methods *depositpay* and *paybill*, and give the code two concurrent transactions. Suppose the pseudocode for *depositpay* describes a transaction that is comprised of several actions. First an en-

1. create the depositpay environment
2. read account
3. read amount
4. compute sum
5. write account (in the containing environment)
6. close the depositpay environment

Figure 2-2: The actions of the *depositpay* transaction

vironment is created for the procedure. The data from the *account* and *amount* arguments are read into the method's environment [1]. The sum of the two variables is computed. That sum is written to the address of *account* in the containing environment. Finally the environment is closed. The steps occuring within the environment are as in Figure 2-2. There is an obvious analagous expansion of the pseudocode for the procedure *paybill*.

Suppose company X wants to deposit John's \$800 paycheck in his account, while company Y wants to automatically withdraw John's electric bill of \$100. Assume John's bank account initially has \$600 in it. Think of each operation as a transaction on John's bank account occurring concurrently. If the transactions are isolated we expect John to end up with a balance of \$600+\$800−\$100=\$1300. In the absence of locks, these transactions are not guaranted to be isolated. There exists an ordering of their composite actions such that the effect is not the same as if either all of X occurred before Y or vice versa. For example, the sequence of steps with their associated values in Figure 2-3 demonstates one such ordering. John will be very disappointed to discover he has only \$500 in his account.

With the use of locks we can ensure John will not be surprised the next time he looks at his account statement. Consider the modified procedures *depositpay-isolated* and *paybill-isolated* in Figure 2-4 that make use of the special procedures *lock* and *release*. The locking protocol marks the address of its argument in the containing environment such that no other environment may access that address until the mark is removed. A transaction attempting to lock an address that is already marked must either wait for the lock to be released or perform some alternative computation.

Action	Value
1. read account_X	600
2. read amount_X	800
3. compute sum_X	1400
4. read account_Y	600
5. write account_X	1400
6. read amount_Y	100
7. compute difference_Y	500
8. write account_Y	500

Figure 2-3: Sequence of parallel actions violating the isolation constraint

```
depositpay-isolated:   procedure(account, amount);
                       lock(account);
                       account = account + amount;
                       release(account);
                       return;

paybill-isolated:      procedure(account, amount);
                       lock(account);
                       account = account − amount;
                       release(account);
                       return;
```

Figure 2-4: Locking in the *depositpay-isolated* and *paybill-isolated* transactions

In this case we just wait until the lock is released. The mark remains until the transaction that made the mark releases it. The new isolated procedures prevent the former problem of transactions reading and writing to the same location. John is guaranteed, as long as the transactions commit, to have $1300 in his account.

The locking protocol can be a source of many pitfalls. It is important that the designer of a system undertands the locking protocols of the modules he or she is creating or using. Particular attention should be given to the case where transactions reach a state where none progress because all are waiting for a lock on a particular resource. We use the term deadlock to refer to such a situation [14]. Several possible implementations can be constructed to avoid this problem.

Some common isolation errors permit one transaction to observe the composite nature of other transactions. Dirty reads allow one transaction to see uncommitted changes from another parallel transaction. Nonrepeatable reads allow a transaction to observe changes made when other transactions commit. Phantom reads involve being able to see changes that could increase the size of the results of a database query [10, 29]. Suppose one transaction executes a query. Then another transaction commits a change that adds contents that also fit the query. If the first transaction executes the same query again and the new results include the newly added contents, we have witnessed a phantom read.

Achieving isolation is an important concern in any transactional system. A mechanism to read and write data atomically and durably does not prevent the programmer from violating the ACID design requirements. One example is that of using locks to isolate the *depositpay* and *paybill* procedures. The triple store can be used to achieve sound transactions. Database isolation level settings provide isolation for physical data. However, since multiple transactions can run concurrently in Haystack, programmers must be careful not to violate any isolation constraints when sharing application data that pertains to triple store data.

2.2.3 JDBC: A Transactional API

The `JDBCStore` implementation uses the Java DataBase Connectivity (JDBC) API. JDBC provides programmers with a tool to connect to virtually any data source that uses a tabular structure. These data sources may be in the form of flat files, spreadsheets, or databases. The API uses the Structured Query Language (SQL) language to communicate with the database [25]. Once a connection has been established with the data source, the programmer implements *put* and *get* operations to physical storage with a series of SQL statements. *Put* and *get* write and read data values, respectively. Before being able to use the *put* and *get* methods, SQL requires the programmer to create the tables in which SQL objects will be stored if they do not already exist. In this section we present an introduction to relational databases, some alternatives to relational databases, a description of some of the technologies specific to the JDBC API, and a database named PostgreSQL [13, 27] that we currently use as our JDBC backend.

Relational Databases

The Structured Query Language (SQL) provides a language for communicating with a Relational Database Management System (RDBMS). The `JDBCStore` implementation writes its *put* and *get* methods as SQL statements. The fundamental structure in a SQL database is a two dimensional table. Columns represent attributes, and rows contain values for some or all of the columns. This flat table structure alone, however, is not very useful. The power of relational databases comes from combining the many tables that comprise the database into results for database *get* operations, or queries. *Get* operations may be combined through conjunction and/or disjunction or hierarchically to perform complex *put* and *get* operations.

A SQL query may combine columns from multiple tables. Whenever a boolean test clause in a query involves two columns from different tables we call the operation a *join* operation. Querying multiple tables in the absence of any *join* constraints returns the Cartesian product of the rows from each that satisfy the other query

RDF_Assertions

assertion_id	subject	predicate	object
1	flowers	have	petals
2	flowers	bloom	in-the-spring
3	my-garden	has-many	flowers
4	the-grass	is	green

Figure 2-5: A Sample SQL Table

constraints [7]. *Join* operations embody the constraints necessary to describe the subset of the Cartesian product desired in a query result set. A less restrictive type of *join*, called an *outer join*, does not block a row from selection when the value in one of the tables is empty [11].

The simplest operations included in the SQL grammar are `CREATE TABLE`, `INSERT`, `SELECT`, `UPDATE`, `DELETE`, and `DROP TABLE`. Users can't do anything with an empty database that has no tables, so the first step is to execute a `CREATE TABLE` statement that defines the column structure with a SQL type and a name for each column. Often a primary key is designated to uniquely identify a row in the table. Suppose we are defining a table named *RDF_Assertions* that has as columns an *assertion_id* primary key, a *subject*, a *predicate*, and an *object*, as in Figure 2-5. The value of the *assertion_id* primary key attribute in the first row is 1, and is hence a unique identifier for the row. The *RDF_Assertions* table will not accept addition of another row with an *assertion_id* of 1. Once the table structure has been defined, we can begin to add data to our database. The `INSERT` operation adds rows to the table, specifying values for the columns. It is one of the SQL *put* methods. The statement `INSERT INTO RDF_Assertions VALUES('1', 'flowers', 'have', 'petals')` creates the first row of Figure 2-5. The `SELECT` operation can then be used to generate database queries as mentioned earlier. It is a SQL *get* method. The operation `SELECT * FROM RDF_Assertions WHERE assertion_id = '1'` returns the first row of the *RDF_Assertions* table. The `*` character signifies selection of every column from the table. The `UPDATE` operation can modify a row or rows in a table.

SQL has the power to do a *get* across rows and *put* specified column changes to selected rows. There `WHERE` clause again defines constraints as in the `SELECT` operation. The statement `UPDATE RDF_Assertions SET object = 'in-the-summer' WHERE predicate = 'bloom'` updates the *object* column of the second row of the table in Figure 2-5 to be the value "in-the-summer". The `DELETE` operation allows removal of a row or rows from a table, and is similarly qualified by a set of constraints. `DELETE` is an operation that combines a *get* across rows and a special type of *put* that removes the selected rows from the table. Finally, if a database table becomes useless, the `DROP TABLE` statement can be used to expunge every row in the table and the table itself.

Because the `CREATE TABLE`, `INSERT`, and `SELECT` statements can only add information to the database, the destructive impact of a malformed statement is often not very severe. Mistakes in `UPDATE`, `DELETE`, and `DROP TABLE` statements have a far greater potential for ruining the consistency of an application that uses SQL [12]. For this reason SQL databases frequently have the ability to restrict certain users from performing the different types of statements.

The Object-Oriented Argument

An alternative to relational databases is the object-oriented database [30]. Object-oriented databases can store arbitrary data structures, making them conceptually easier to understand than relational databases. They are specialized for writing these data structures to disk and don't have the query performance enhancements of relational databases. The *put*s and *get*s Haystack data requires are very uniform since only a small part of the application actually controls the data model. Haystack users will also frequently perform queries which touch this data model. Object-oriented databases are therefore less desirable Haystack data storage solutions.

Object-Relational Databases

Object-relational databases combine the ease of modeling the complex data structures of an object-oriented application with the ability to express varied relational SQL-like

queries without much coding overhead. They are useful for complex object-oriented applications that would also benefit from a dynamic set of query functionality. However, these databases are typically not optimized for transaction processing of the frequent queries we expect from Haystack [30]. Our data is also very regular, so we don't need the complex data structures that object-relational databases support.

The JDBC API

The triple store uses the JDBC API to access its database. JDBC is a standard package in the Java programming package [32]. This package gives the programmer an API for creating a SQL database. The API boasts seamless database connectivity for a number of the database management systems available. It also provides a mechanism, for those databases that support it, to serve as the basis for a transaction management system.

We want Haystack to be able to run on as many platforms as possible while minimizing user overhead. Our choice of third-party databases should not require that the user have a particular database. One of the reasons for choosing to use JDBC in our triple store implementation is its ability to use a myriad of data storage applications and formats. We use JDBC with PostgreSQL in the hopes that the transition to another database will run smoothly.

A major weakness of JDBC is its inability to perform the basic administrative functions necessary to set up the database for use. Perhaps this partially identifies a fault in conventional system design. JDBC reduces the problem of using an arbitrary database to the problem of installing and configuring that specific database properly and configuring the Java Virtual Machine (VM) to connect to it.

To perform the neccessary database installation and setup, we must go through the operating system. Setup procedures either require writing system dependent Haystack install scripts to configure and run the database server, or asking the user to perform the install. Each scenario violates a different Haystack design goal. The former violates the idea that Haystack should run seamlessly on any platform. The latter violates the idea that user overhead should be minimal.

Once a JDBC connection can be created, we need to establish our transaction protocol. The `JDBCStore` can dynamically load a database-specific JDBC driver and establish a connection. The connection can be configured to allow multiple SQL statements in a single transaction. It can also be configured for the desired transaction isolation level. There are several settings for this, but the one we are interested in is referred to as `TRANSACTION_SERIALIZABLE`. This level provides the most restrictive set of rules for isolation between transactions. It prevents dirty reads, nonrepeatable reads, and phantom reads. Each of these events represents an act that would break the ACID properties of a transaction. There are performance costs when using this restrictive isolation level [29]. The underlying database probably implements some form of locking strategy to ensure isolation of the values transactions access. But the benefits of ACID compliance are enough to justify these costs.

Once we have made our connection and set up our transaction strategy, we can begin executing SQL commands with interfaces from the `java.sql` package [32]. A transaction is implicitly open on our connection at this point. To run another transaction at the same time, we have to have another connection. Individual `Statement`s are obtained from a `Connection` object. We may execute any valid SQL command on a `Statement`. If the `Statement` was a query, we get back a `ResultSet` object that allows us to iterate through the results. A `Statement` works well for datatypes that can be easily included in a string and statements that are only executed once. The `PreparedStatement` allows us to optimize performance when similar SQL will be executed repeatedly or with slight parameter changes. The parameters can then be set to values of any SQL type. There is a mapping from Java object types to SQL data types provided. We can set values on a `PreparedStatement`, execute one query, change the values of the `PreparedStatement`, and execute that query. After we are done executing our series of `Statement`s and/or `PreparedStatement`s, we simply call a commit on the connection. If something goes wrong at any point, we instead call an abort on the connection. After the commit or abort call returns, the connection implicitly begins its next transaction [25, 29, 32].

PostgreSQL

The triple store was implemented and tested using the database PostgreSQL. It is an open source database which installs on almost any modern Unix-compatible, Windows NT-based, or Windows 2000-based operating system. It is an Object-Relational Database Management System [13]. The PostgreSQL community claims to support almost all SQL constructs, including subselects, transactions, and user-defined types and functions. The PostgreSQL JDBC driver claims to support the standard JDBC interface [32], and the community advises users to consult the Sun documentation [25, 32]. Supplementary documentation, tutorials, and support may also be found at the PostgreSQL Web site [27].

While PostgreSQL is an object-relational database, our interface is restricted by our decision not to require a particular Haystack database backend. We only use the standard SQL grammar supported by the JDBC interface, even though PostgreSQL offers some extensions.

2.3 Getting Haystack to Understand

To help Haystack describe its data, the triple store is designed to store a flexible Resource Description Framework (RDF) model [15]. Traditional applications write down their data using database tables that have no meaning unless we can understand the structure and interpretation of the tables. RDF provides a way for an application to write down different data as descriptions. These descriptions give meaning to the application's internal structure and interpretation of data.

The intent is that when Haystack migrates its data model to use the triple store, we will define a Haystack RDF model that is compatible with Web technologies. We hope RDF will be flexible enough to make the transition relatively easy. This uniform RDF model will make pieces of a Haystack's state more easily portable to other Haystacks. We expect that collaborating Haystacks will enhance a user's search by asking other Haystacks for help with searches when appropriate. In the future Haystack services may exist to incorporate and understand information from foreign RDF applications.

2.3.1 An Introduction to RDF

RDF was created to make information available on the World Wide Web machine-understandable [26]. RDF represents information in the form of an assertion that describes some relationship between data objects. An assertion's data objects, or resources, can be any entity, digital or not. The premise is that the metadata that describes a resource can provide an understanding of the resource. The metadata, which is itself syntactically indistinguishable from the data, gives context to the resource. From this context a machine can discern meaning.

RDF processors can automate the task of processing the information in an application's RDF model. These processors read the contents of a model and trace the model's meaning back to something they can understand. A foreign RDF model may refer to schemata that define the types it uses to describe its data. The generic concept of a type or category is called a class in RDF [15]. A schema may in turn refer to classes defined in other schemata. All schemata may be traced back to the RDF Schema Specification 1.0 [15]. If an RDF processor encounters metadata it cannot understand, it can trace the class hierarchy back to this schema.

XML is a syntax one can use to write down an RDF model. It is a well supported syntax that provides information in a machine-readable form [5]. Future Haystack research may include adapting the XML parsing service to read and write to the triple store. The triple store design evolved from a combination of both the current Haystack Data Model and the similar RDF model. Using XML to write this model is beyond the scope of this work. XML is described in detail at the World Wide Web Consortium Web site [5, 14, 16]. An XML formal grammar that may be helpful in understanding RDF terminology is provided for reference in Appendix A. The XML serialization of the RDF Schema Specification 1.0 found in Appendix B may also be useful. These appendixes read much like HTML and may still be somewhat useful even if the reader is not familiar with XML.

2.3.2 RDF Basics

The most basic RDF expressions involve resources and properties [26]. A resource is a node in an RDF model that can be identified by a unique Universal Resource Indicator (URI). URIs are typically character strings that identify a resource by concatenating a protocol, a scope or namespace, and a scope-specific identifier. A property is a relation that embodies some aspect, characteristic, or attribute of a resource and maps to some value. In object-oriented terminology, resources are to objects as properties are to instances. The properties of a resource give the resource its context and meaning. A collection of RDF expressions can be resolved in an RDF data model or context for representation and comparison. "Two RDF expressions are equivalent if and only if their data model respresentations are the same" [26].

An RDF statement, or assertion, is an ordered RDF triple that contains a resource, a property of that resource, and a value for that property. Hereafter these three statement elements will be referred to as the assertion's subject, predicate, and object, respectively. The subject is any resource about which we assert a particular property. The predicate is a resource that may have other properties attached to it to clarify its meaning and/or constraints on its subject and object. The object may be either a resource or literal.

It is unclear to the author exactly what an RDF literal is from the literature. A literal is primitive data that is not processed by an RDF processor. Literals may appear as XML expressions [26]. For our purposes in the triple store we accept arbitrary bit string literals. An object can be a literal while a subject cannot be a literal. It is unclear whether or not a predicate can be a literal.

The triple store writes its RDF model with statements that can have only resources as subject, predicate, and object. To store an assertion that contains a literal, we first store the literal, and instead use the internal resource that refers to the literal in the statement. We note that this requirement merely describes a particular way to write statements with literals and does not break the RDF specifications.

In order to represent more complex RDF expressions such as statements about

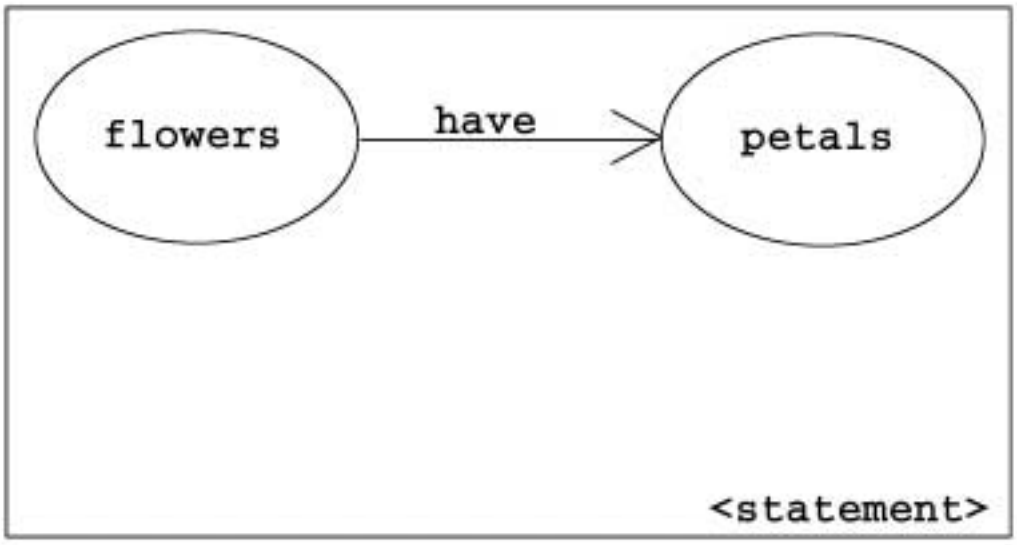

Figure 2-6: A Simple RDF Model

statements, we have an RDF type system. A type is a specific property in the RDF model [26] whose value describes the type of its subject.

A container is a collection of resources. Containers can express assertions about mathematical sets. Properties can be attached to a container of authors to describe something about all the authors of Haystack documents.

We can express higher order descriptions by writing statements about statements. In order to refer to a statement itself as the subject of another statement we must introduce a resource in our model which represents the entire statement. The process of creating this new resource is called reification [26]. Haystack uses a lot of higher order metadata, so we reify every statement in the triple store.

2.3.3 An RDF Example

Consider the case where we have three resources called "flowers", "have", and "petals". To express the description "flowers have petals", we introduce the assertion with appropriate subject, predicate, and object to produce Figure 2-6. The statement itself is labelled in the figure as "<statement>".

We use nodes to represent resources and labelled arcs to represent properties. Note

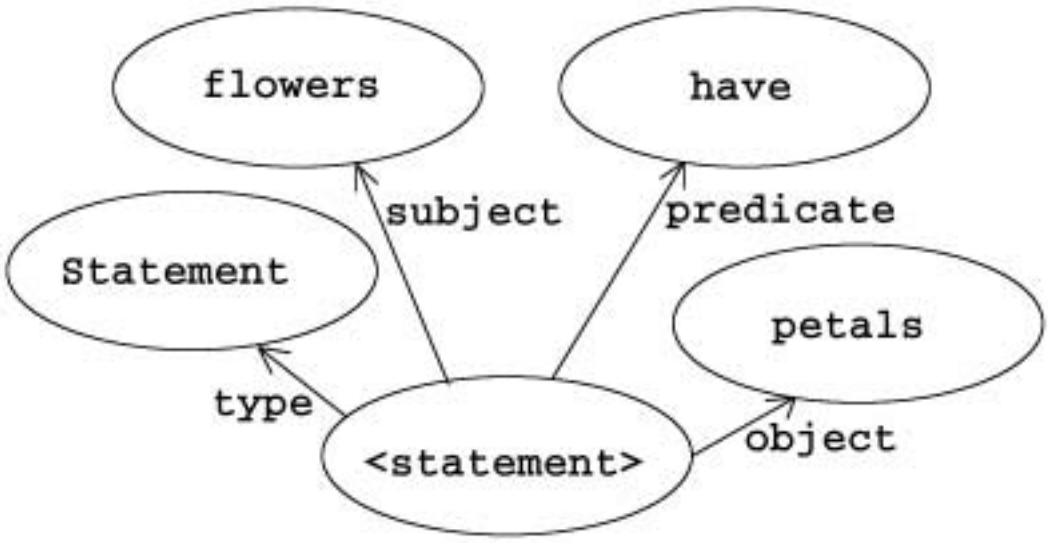

Figure 2-7: An RDF Model for a Reified Statement

that the labels on the arcs can themselves be considered resources, since properties are resources. We use abbreviated names at this point and consider all identifying URIs to be in a uniform local scope.

Now consider the case where we want to attach something to our previous "<statement>". We want to be able to express a statement about our "<statement>". We perform the reification to obtain Figure 2-7 that has a node for "<statement>". The "<statement>" resource has the (*attribute, value*) pairs {(*type, Statement*), (*subject, flowers*), (*predicate, have*), (*object, petals*)} created by the reification. The reification still expresses the assertion "petals have flowers".

Suppose we now want to say "Alyssa P. Hacker says flowers have petals". To capture this statement in our model we attach an assertion with the original "<statement>" as the subject. We use the property "assertedBy" to embody the concept of one entity saying a certain statement holds. Figure 2-8 shows the resulting model. The additional "assertedBy" property with value "Alyssa-P-Hacker" has been added to the "<statement>", giving it more context and meaning. The triple store interface automatically reifies all statements by giving each one a new URI.

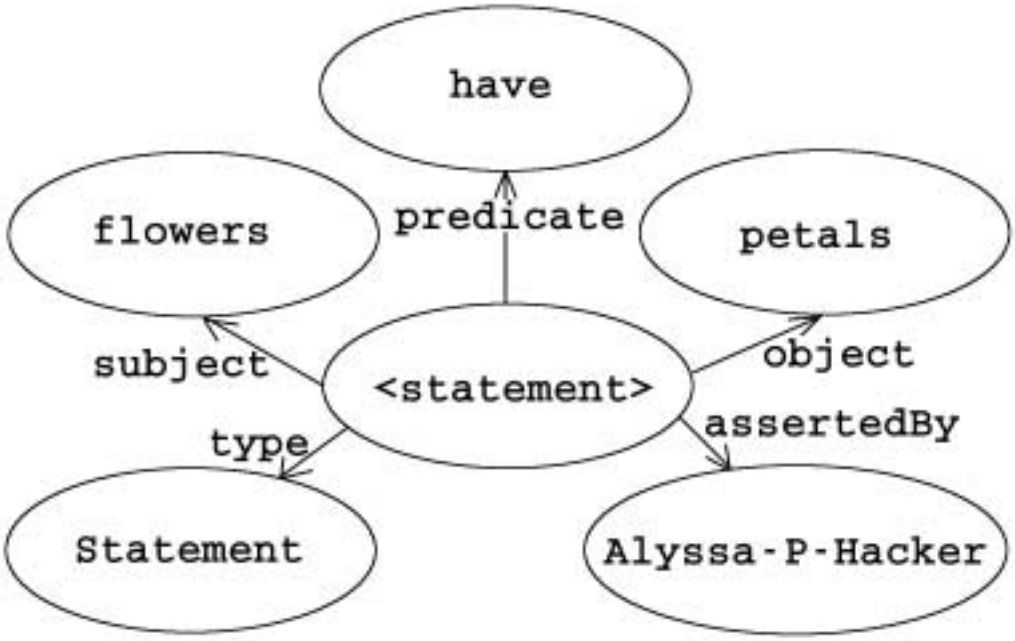

Figure 2-8: An RDF Model for a Statement About a Statement

2.3.4 Adding Scope to RDF Resources

We now relax our constraint that every resource in our RDF model must be in local scope. We allow our model to refer to other RDF models. To accomplish this task we use RDF namespaces. If an RDF model is written in XML, the model's RDF namespace is the same as the XML namespace that describes the model [14]. Some areas such as the finance domain already have well-understood and widely used XML vocabularies. Instead of creating their own namespaces, RDF models should use these namespaces wherever appropriate.

When examining models, RDF processors must be able to recognize which resources are to be processed in the local scope and which refer to a foreign scope. Recall that URIs are universal versions of all names used in an RDF model that extend beyond their local model. A resource that is outside of local scope has a URI that identifies the namespace to be consulted to resolve its meaning [14]. Since an RDF namespace is a special kind of XML namespace it is referred to by the variable prefix *xmlns*.

We return to the example statement, "Alyssa P. Hacker says flowers have

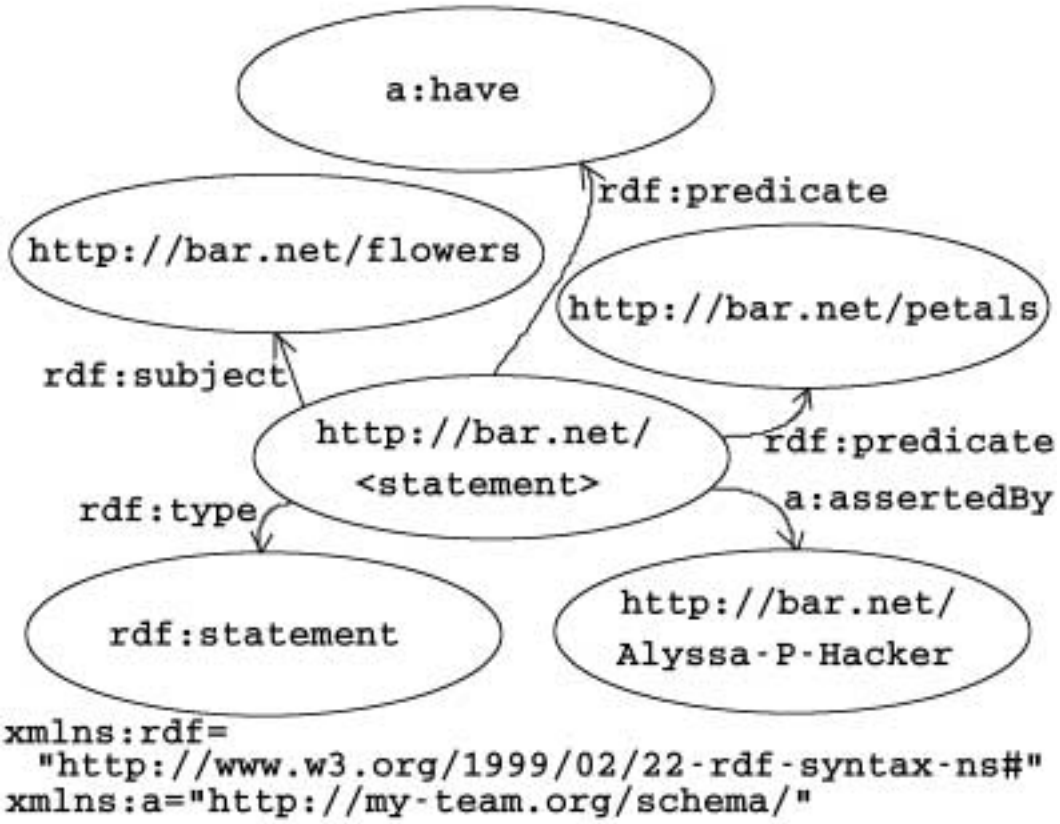

Figure 2-9: An RDF Model With Namespaces Identified

petals", and modify it slightly to allow to namespaces outside of the local scope. We refer to the basic resources of the RDF Model and Syntax [26] with *xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"*. This namespace defines the RDF resources "type", "subject", "predicate", and "object". Suppose further we have a namespace *xmlns:a="http://my-team.org/schema/"* that describes the "assertedBy" and "have" properties. A final namespace clarification might identify the data repository "http://bar.net/" in which we define the resources "Alyssa-P-Hacker", "flowers", "petals", and "<statement>". We now have Figure 2-9, which shows a universally readable version of the example from Figure 2-8. The higher order statement can now be read and understood by a remote machine.

2.3.5 The RDF Type Hierarchy

The RDF type system will become important in follow up work to this thesis, when Haystack starts using the triple store. It is very similar to the type systems of object-oriented programming languages and is described in the RDF Schema Specification 1.0 in the *xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"* namespace [15]. Recall the namespace *xmlns:rdf* presented ealier that defines some of the basic RDF resources [26]. The author knows of no enforcement of the rules given in these schemata except possibly the fact that models that violate them may not be considered valid RDF models at all. In this case, RDF processors would be explicitly forbidden to process them.

The basics resources of the RDF Schema 1.0 are shown in Figure 2-10. The root node is the resource "rdfs:Resource". Two principal subclasses of this resource are "rdfs:Class" and "rdf:Property". The properties "rdf:type" and "rdfs:subClassOf" are used in the figure to describe the type and class hierarchy of a resource. Any class defined in RDF has "rdf:type" "rdfs:Class", including the resource "rdfs:Class" itself. A statement with predicate "rdfs:subClassOf" specifies that the subject is a subset of the object. Properties may be appended to the model to describe existing resources. The "rdfs:subPropertyOf" predicate is used in RDF assertions to declare that the subject is a specialization of the object.

For an arbitrary resource, the type property is used to signify that the resource is an instance of the specified class object. An instance of a class has all the characteristics expected of members of that class. RDF permits a resource to be an instance of many classes. The subClassOf property is a transitive property. To prevent looping in the model, a class can never be its own subclass or a subclass of any of its own subclasses. Transitivity also applies to the subPropertyOf property.

There are also several extensions to the basic property and type system. Properties can be added to other properties to further describe where they may be legally applied. For example, a restriction could require that a certain property may be applied to at most one subject. One extension to the RDF Schema 1.0 is the DAML

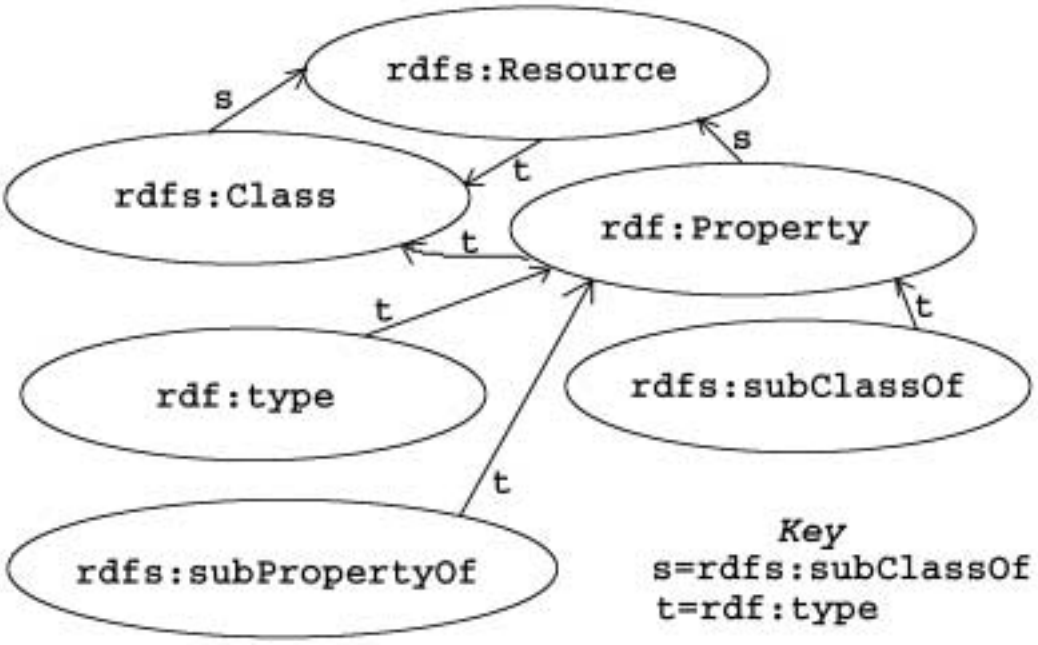

Figure 2-10: The Principal Resources of the RDF Type Hierarchy

Ontology [22]. The DAML Ontology introduces several mathematical qualifiers to the model. It describes set concepts such as cardinality, disjoint, disjoint union, intersect, equivalence, membership, and complement. The instances of a class comprise the set defined by a given class. The DAML Ontology also defines properties about relations, such as uniqueness, inversion, and transitivity.

We expect that Haystack will be translated into an RDF model using the tools provided by the RDF Schema and DAML Ontology. As this type system is very similar to object-oriented type systems like Java, the migration should be somewhat straightforward.

In the next chapter we describe the Haystack Data Model in more detail and discover that Haystack already uses metadata relations to express a lot of its state. The next chapter also presents some of the processes which run in the Haystack Service Model to help with search tasks. We see that it is important to protect data from these services to ensure the robustness of the triple store. Some of these services are useful to the triple store and had to be modified for use outside the root server.

Chapter 3

The Haystack Personalized IR Tool

Many parts of the existing Haystack system had profound impact on the creation of the triple store. Minimizing user overhead with flexible modules is an important part of both Haystack and the triple store.

The Haystack Data Model (HDM) that was already in place used a lot of metadata. As this data model preceded the notion of RDF, it instead used a less uniform storage model for its relations. We hope that the triple store will enhance the HDM by allowing it to describe itself with RDF assertions.

The Haystack Service Model (HSM) was important to the triple store as well. The Haystack Trust Model of Chapter 4 has emerged to allow the triple store to make use of legacy services. The problem with the HSM's existing trust mechanism was that it permitted at most one instance of a service to run at a time while requiring that all services talk to the root server. Since the triple store cannot break existing code, the Haystack Trust Model does not affect the operation of the Haystack Service Model presented here.

There are also problems with transaction management and robustness in the existing data storage services. The triple store hopes to provide a more robust interface with working transactions.

We begin with overviews of the Haystack system and some Haystack goals to help the reader get acquainted with the project.

3.1 What Haystack Does

The problem with current information retrieval tools is that their structure often doesn't suit the user's needs. Haystack is an effort to exploit the potential of information retrieval for the user.

As users add more and more information to their digital information space, organizational structure becomes increasingly difficult to manage. Conventional file systems require too much structure, and their files and directories contain almost no tracking data to describe their content and layout. File systems make logical interpretation difficult, especially when trying to understand the filesystems of friends or colleagues. The solution seems even more evasive when we look at all the data available on the Internet. The structure of a Haystack information space is more flexible and can contain metadata to describe its layout.

3.2 Haystack Goals

Here we visit, clarify, and modify the legacy design goals [3] of the Haystack project in the context of this thesis. Haystack should:

- include persistent and dependable modules so that users can rely on it to store information.
- provide easy customization.
- allow distributed and dynamically loadable utilities and services. Utilities should be reusable in other modules whenever possible.
- be designed with an eye towards permitting inter-Haystack collaboration.
- give users an easy-to-understand querying mechanism that provides accurate IR functions.
- provide personalized query results.

- provide the ability for users to both explicitly and implicitly annotate their information space.

- learn from the past, and adapt to the user's changing information needs.

Since the triple store is a low-level module it is really only concerned with the first five points of persistence, easy customization, reusable and loadable utilities, Haystack collaboration, and a remote query command line.

For users to start relying on Haystack they must be assured that Haystack is persistent. At the lowest level of persistence are data storage layers such as the triple store. The triple store requires little user overhead beyond the administrative tasks of setting up a database and providing a pointer to the right JDBC driver. The `JDBCStore` also incorporates reusable and dynamically loadable communication utilities. The uniform methods of the RDF triple store will hopefully make describing Haystack data to another Haystack easier. The triple store module also contains a skeleton for invoking *put* and *get* methods directly to the module from a remote command line.

3.3 Haystack: The User Perspective

To use any of the Haystack interfaces, I first run a Haystack server with default preferences. Users can choose from a command line [2, 3], a Web client, or a Java Graphical User Interface (GUI) [21]. The remote command line and Web client require password authentication. To these interfaces we add the skeleton of a remote triple store command line that also requires password authentication[1].

I begin by bringing information into my corpus of data through a process we call archiving. Suppose I tell Haystack to archive the MIT homepage at "http://web.mit.edu/". The request tells Haystack to store the contents at that URL, and some of the links that it provides, into Haystack. I can now perform a query over my information space with the term "science". I am given a list of resources, called

[1]The triple store and root server currently share the same authentication.

HDM		RDF
Straw	≡	Resource
HaystackID	≡	URI
Tie	≡	Property
(Straw, Tie, Straw)	≡	(Subject, Predicate, Object)
	≡	Statement or Assertion
Needle	≡	Literal
Bale	≡	Container

Figure 3-1: Similarities between the HDM and RDF

straws, representing "science" in my Haystack. I can follow a link, called a tie, to the page labelled "Massachusetts Institute of Technology" and find a text description stating "MIT is devoted to the advancement of knowledge and education of students in areas that contribute to or prosper in an environment of science and technology."

The following sections look closer at parts of Haystack. We discuss the existing pieces that have had an impact on triple store design.

3.4 The Haystack Data Model

Haystacks contain straws, ties, needles, and bales. These objects form the basis of the labelled, directed graph, or digraph, through which a user's searches navigate. All HDM objects are of type straw and have a unique HaystackID to identify them.

HDM objects are very similar to RDF objects. The reader may wish to refer to Figure 3-1 as a guide to similarities between the HDM and RDF.

3.4.1 Straws

Straws are the basic nodes in the HDM digraph [3]. Everything in this digraph is a straw, including the edges connecting the nodes. Each straw subclass has a type that provides semantic information about the straw. A type can be used as a clarification during search [2] if, for instance, the user has requested only PostScript documents. Straws have pointers to their forward and backward edges in the graph.

A straw object is very similar to an RDF resource. All HDM objects are straws, and all RDF objects are resources. All straws have a unique HaystackID, and all RDF resources have a unique URI. The triple store can be used to represent both an HDM model and an RDF model.

3.4.2 Ties

Ties are the edges of the HDM digraph. They are special types of straws that connect two straws in a directed fashion. A tie has exactly one special backward and forward link. The backward pointer indicates the source about which a tie is created. The forward pointer indicates the value of the tie for that source. Additional information, in the form of more ties, can be added to a tie to expand its semantic meaning. To add an edge to a tie we transform the edge (tie) into a node and create a special edge to its source and its value. Ties are represented as nodes with special unlabelled forward and backward edges when we resolve our statement "flowers have petals" in the HDM in Figure 3-2.

A tie is similar to an RDF property. A tie acts on a source connecting it to its value, and an RDF property acts on a subject to provide a particular value. If we take a tie together with its source and sink straws we have the equivalent of an RDF subject, predicate, and object triple. This triple is an RDF statement or assertion. The process of transforming a tie from an edge to a node in the HDM digraph of Figure 3-2 serves the same purposes as reification of an RDF statement. Reification allows us to attach statements to statements.

3.4.3 Needles

Needles are straws that contain a data element, and take the form of locations, file types, bodies, and text strings [3]. They are the raw bit strings in the HDM. One type of needle is the bits of a document. Metadata surrounding the contents of a needle usually describes information like how that document should be displayed to the user and the term frequency measurements calculated for retrieval.

A needle is similar to an RDF literal. Recall that for the purposes of the triple store a literal is just a collection of bits. Raw bits in both models are essentially unprocessed, but in Haystack a particular service may read the contents of a needle and perform some task in response. The meaning of both a needle and an RDF literal is described by its attached assertions.

To *put* an RDF statement that contains a literal in the triple store, we must use a URI that refers to that literal. Needles are the reason for the indirection. Since we always have a HaystackID that refers to a needle in the HDM, we require that we always have a URI that refers to a literal before we use it in the triple store. As mentioned earlier it was also unclear to the author exactly when literals are permitted in an RDF statement. Requiring a URI for a literal makes it so that we always put non-literals in our statements, and capture the same meaning. The property "hasBits" is a part of our Haystack RDF model that maps a resource to a literal. Assertions containing this property express the concept that a subject has a bit string representation corresponding to the appropriate literal.

3.4.4 Bales

Bales are straws that represent clusters of related resources for a document. A bale is used to encapsulate the idea of a collection in Haystack. Usually a particular document type will have a bale of ties that are commonly associated with it, such as "author" and "creation date". Bales are used for the efficiency gains of grouping resource clusters in the HDM [2].

A bale is similar to an RDF container. Bales have other straws as members and can be defined for a certain purpose, and containers have other resources as members and allow properties to be added to the entire collection. Bales will be deprecated by indroducing an appropriate type tie to express the idea of a container.

3.4.5 Navigating the Model

Once we have stored things in our Haystack, how do we get around? When a document is archived in Haystack, a small web of semantic information is attached to the document. The Haystack program has access to the forward and backward links of a straw and can follow them to examine this web. All of the semantic information surrounding a document can be found in the connected component of the HDM that contains the straw that represents the document. While executing a query, Haystack can search through the document's metadata to determine if it is relevant to the user's query. Contextual clues litter the data model, giving Haystack the ability to understand more about its resources. If Haystack decides a resource is relevant, it is returned to the user in the query results. From this resource, the user can follow the links of the data model just as the automated query processing has[2].

We return to the example in which we modelled the statement "flowers have petals." This same statement could appear as a portion of the Haystack Data Model, as shown in Figure 3-2. Here we see a network of straws. Note that each straw, whether it be a straw, tie, or needle, has an ID associated with it. This is the HaystackID. Note that a straw, including the one with "ID: 1", can have multiple "back" or "forward" links. A tie has a special "back" pointer and a special "forward" pointer identifying the two resources it connects. A needle may be resolved directly to a string literal, or any other arbitrary stream of bits, via the appropriate *get* method.

3.5 The Haystack Service Model

Haystack is driven by its services. Each service has a service name associated with it that provides a semantic description of it's function. There can be at most one service with a given name running inside Haystack at any time. Services depend on the existence of a root server.

In the past the root server was synonymous with the entire Haystack program.

[2]The user's view of the data model may be a subset of the view that query services can traverse.

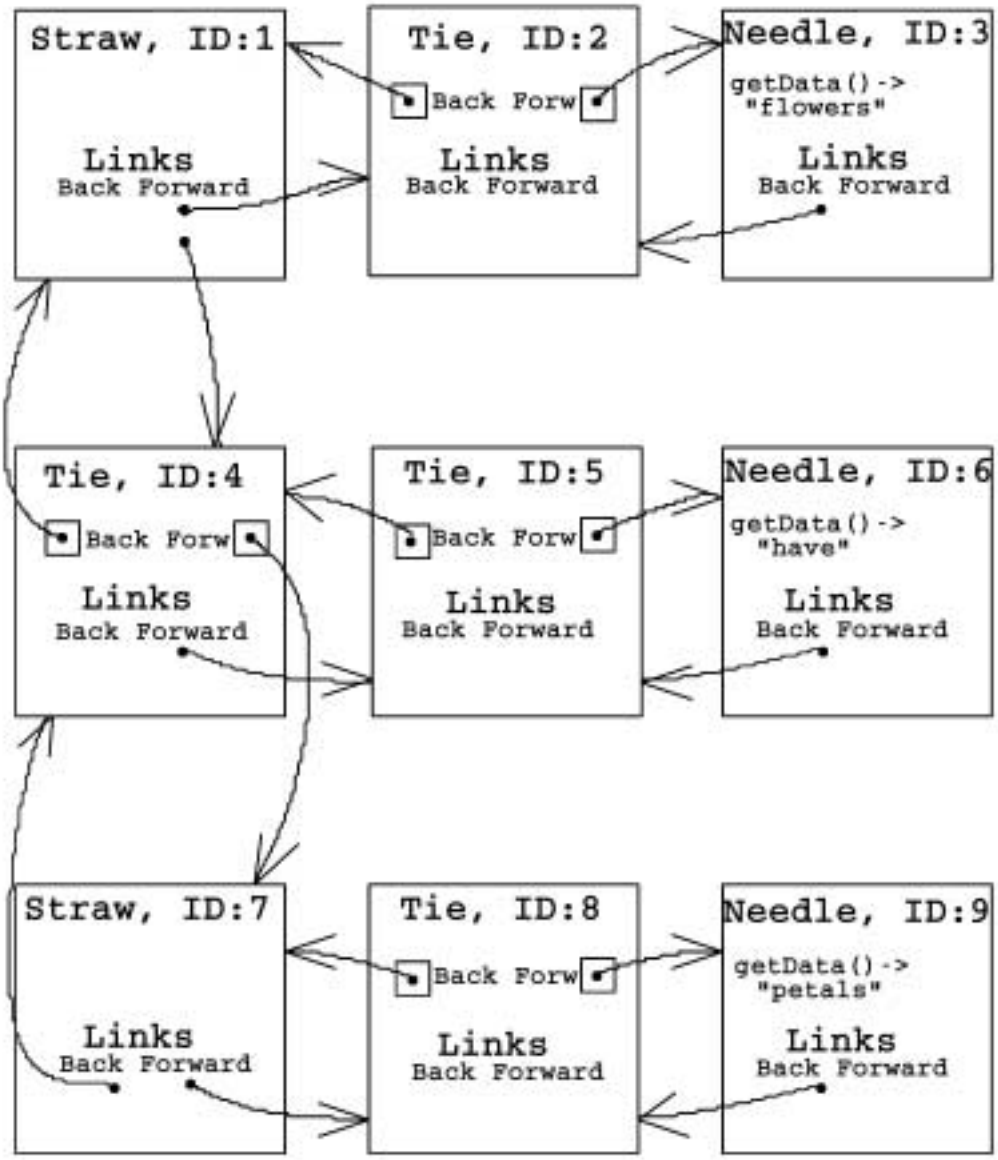

Figure 3-2: The HDM for "flowers have petals."

We now have a triple store module that runs with or without a root server. When we removed the root server for the first time, services became useless. The triple store uses modified versions of some of the services that are described in this chapter.

We begin with a definition of `HsService` and discuss some other relevant services.

3.5.1 HsService

`HsService` is the superclass of every Haystack service. The `HsService` constructor is given a service name identifier made up of the short name, package, version, and creator of the service. The short name identifies the service's Haystack type. The package identifies the service's module. The version number corresponds to the different revisions of a particular service's class. We envision the dynamic introduction of new versions in Haystack that can interact with older versions or other services. The creator identifies the entity that created the service. The entire service name tuple uniquely identifies an instance of a service in Haystack. Instances of `HsService` should be instantiated, intialized for use, and closed for shutdown.

Because we cannot use services outside the root server, we have created utilities that are more flexible than services. Previously, almost everything added to Haystack was a service that extended `HsService`. The problem was that these services became too tightly coupled with the singleton root server running in Haystack. Exactly one instance of a particular service may run in the entire Haystack program, and that service can only run in the root server module. We wanted to use some services in the triple store, but could not. Utilities provide a solution.

3.5.2 Core Services

The core services include the root server, the name service, and the config service [3]. Other core services include loggers, counters, and caches. The assumption is that every service can use any of the core services since every root server starts them.

The Root Service

As mentioned earlier, the root server is the primary driver for the Haystack IR tool. The root server loads in a dynamically configurable set of services according to the config service. The root server's initialization method initializes all services including those dynamically registered with the name service.

The root server is an access point for services to obtain pointers to talk to other registered services. Access is granted through static Java methods. This means that no matter how many threads or services there are, they all receive the exact same pointers when they use the static root server accessor methods. The result is that the entire Haystack process shares the same instances of services. While static accessors have some benefits within the scope of the HSM, this strategy is not useful for other modules that instead use the more flexible and generic utilities.

The Name Service

The access point for utilities in the triple store is the name utility, patterned after the name service. The new Haystack Trust Model uses this non-root server-specific version of the name service. The name service is used for interservice communication. It is essentially a container of semantic pointers to other services. It allows these pointers to be identified by a service name rather than the usual Java-specific pointer.

Access to pointers via service name is more suitable for Haystack's dynamically loadable services. A service name is a semantic description of a service. Services may be of any type but can be accessed via a semantic description. The name service allows us to *describe* services with string identifiers rather than use the stricter rules of typing [2]. In the future, the service name may describe how the service is to be used in terms of the Haystack RDF ontology.

Suppose I want to use the name service to access a query service. In Java I am restricted to using some accessible compiled class. Suppose I need to run an old version of the query service to find results that I was able to find a week ago before upgrading my Haystack. The upgrade replaced the old version, but I can now load

an old version of the service that was stored in my Haystack on disk into the name service. I can reconstruct the object from its bit stream to construct the old version's Java byte code. I can then describe to my name service that I want a pointer to the old query service, and be given a pointer to a type that is not accessible in Java's dictionary.

The name service identification strategy also becomes useful when we want to access an object of unknown type. If more semantic descriptions are added to service names we may be able to ask the name service for a service that can execute a particular type of query. Suppose we get a pointer to service X that is not of type query service but can still handle the task. We can invoke X's specialized query method without knowing its type[3].

Figure 3-3 shows how *Service A* contacts *Service B* through the use of a service name T. The type of the pointer $*B$ returned is different from the type of the semantic description T. To establish a connection between *Service A* and *Service B*, each one must first be registered with the name service. The name service is accessible anywhere by contacting the root server. When *Service A* needs to complete task T, it asks the name service for a service which can help with T. Currently a service name such as T contains only a name, package, creator, and version, any of which may be left out to default values when requesting a service. After Haystack's transformation to an RDF model semantic descriptions of tasks may emerge as part of a service name. *Service A* asks the root server for a pointer to the the service with the description T. The name service searches its tables for a service which can help with T. A pointer to the instance of *Service B* is returned. Now the two services can begin communicating.

Take another look at Figure 3-3 and see that everything in the conventional name-space of the name service is contained within the root server. There is at most one name service.

The triple store uses a similar name utility that serves the same purpose as the name service, except that there can be one for every module and it does not depend on a

[3]Java provides a mechanism called reflection that lets us call the named methods of an unkown class or object.

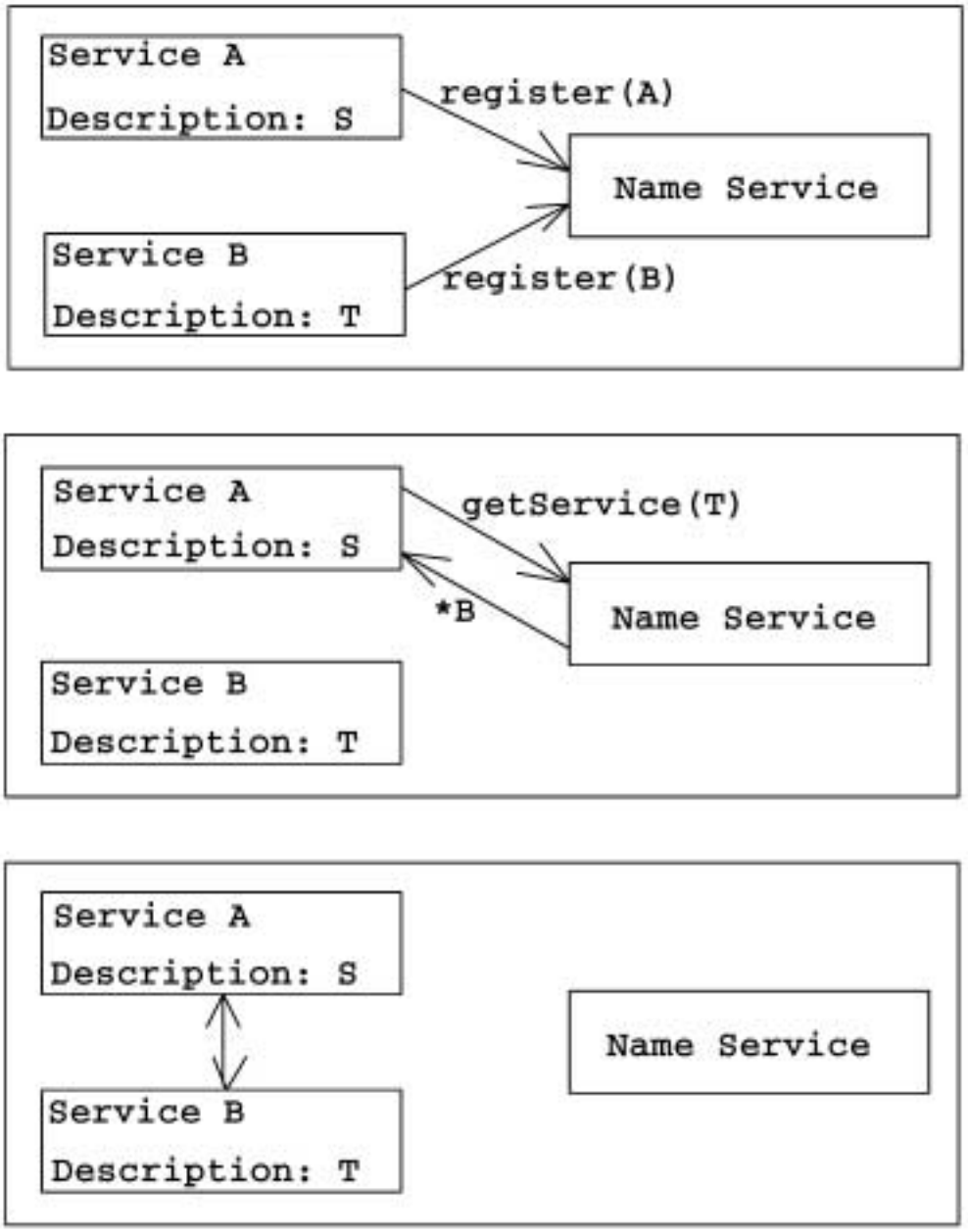

Figure 3-3: Services Communicate via Semantic Descriptions

root server. The name utility handles interutility communication via descriptive utility names. One important difference between the name utility and the name service is that the former does not attempt to persistently save any information on disk about the state of the utilities running within it. It is unclear exactly where an arbitrary name utility should store its utilities lists. Also, the triple store needs no persistent state so there is no need to implement this functionality yet.

The Configuration Service

The triple store uses a generic version of the config service called the config utility that does not require a root server. Both serve the identical purpose of storing the configurations for a user's Haystack. One type of configuration is a list of services that a user wants to load and run in Haystack. Unlike most utilities the global config utility is shared by all modules and contains defaults for all module instances.

3.5.3 Data Model Services

Data model services are those which are permitted to create and store HDM straws, ties, needles, and bales. It is data model services that will eventually be modified to access the triple store interface to physical storage. Access to the triple store module should be limited to data model services, core services, and possibly a few others.

3.5.4 The Communications Module

One of the triple store requirements is that a remote connection directly to the triple store should be possible. In order to use the Haystack communications and security modules for this purpose, they had to be modified. These changes were an unanticipated part of this thesis, but were necessary nonetheless to make service code useable. The Haystack Trust Model emerged in order to be able to use these services in the same manner as they are used in the root server. Whenever possible the actual functionality of parts of the Haystack Trust Model, communications module, and security module was preserved as each was separated from root server dependency.

The communications and security modules support both the root server and the Haystack Trust Model concurrently. The reason is that nothing introduced by this thesis can modify or break the existing root server architecture. The module designs themselves would become much more elegant if the root server were to migrate to the Haystack Trust Model, but such a change is not necessary. For now the communications module fits the service definition if it is instantiated by the root server, but it fits the utility definition if it is instantiated by a triple store.

The communications module includes services[4] that permit processes outside the root server to communicate with the root server. Communication involves sending character string versions of commands across the network to the root server. To use the communications module with the root server these commands must be understood by the `HsCommandAPI` service. If instead we are using the communications module within another module such as the triple store, the commands must be understood by that module's command API.

Remote Service Communication

For services running in a remote process, it is necessary to establish a connection with the root server. The root server has a `MiddlemanServer` listening on a port for connections. The remote client uses a `Middleman` to connect on this remote port. The security package contains objects which facilitate the encryption of communication between middlemen. These security objects are incorporated into the `Middleman` and `MiddlemanServer`, and a password is required from the `Middleman` client. The password and the ensuing client session that follows are encrypted. Packets are encrypted when created, and decrypted on receipt.

The middlemen act as an intermediary for processing the packets, as seen in Figure 3-4. Requests are sent from the client side. The request is packaged up in a packet and sent by the `Middleman` to the `MiddlemanServer` listening on the other end. To the client it seems that it is communicating only with the `Middleman`. A client request

[4] We use the term services since we are talking about the communications module in the context of the root server in this section.

to send a command on the `Middleman` simply returns the response of processing that command.

Behind the scenes a command packet is created, encrypted, and sent to the server. The server receives the command, decrypts it, and executes the command locally. A response packet is created containing the results of local processing, and the new packet is encrypted and sent back to the client.

The Command API

Command APIs serve as the the relay agent to pass messages from the Haystack client and server to their middlemen and back in a typical client-server dialogue. Figure 3-4 traces the path of a request from the client to the server, and the response back. In this example the command line, responding to user input to archive the MIT homepage at "http://web.mit.edu/", begins the dialogue. Time progresses from the upper right of the figure clockwise from client to server and back to client.

We sometimes use the term "virtual" to refer to services in the client and "real" to refer to services on the server side. Client requests are sent to the Virtual Command API. This API asks the client Middleman to send the encrypted command out a socket across the network. The Middleman Server receives the command and decrypts it. The request is passed to the Real Command API. The Real Command API searches its tables of registered commands for one that can deal with the request.

The Real Command API finds and executes the appropriate Archive Command. The Archive Command in turn runs the command on the Root Server, and generates a response for the Real Command API to send back to the client. The response is sent via the Middleman Server wrapped as an encrypted packet across the network. The Middleman receives, decrypts, and returns the response to the Virtual Command API. The Virtual Command API invocation finally returns the response "Archived w/ HaystackID: 3052" to the Command Line and hence the user's terminal. The user is told that the resource has been archived and is free to enter new requests. The actual archiving is queued to minimize the latency of the user's request.

The triple store client uses a similar protocol to talk to the triple store server via

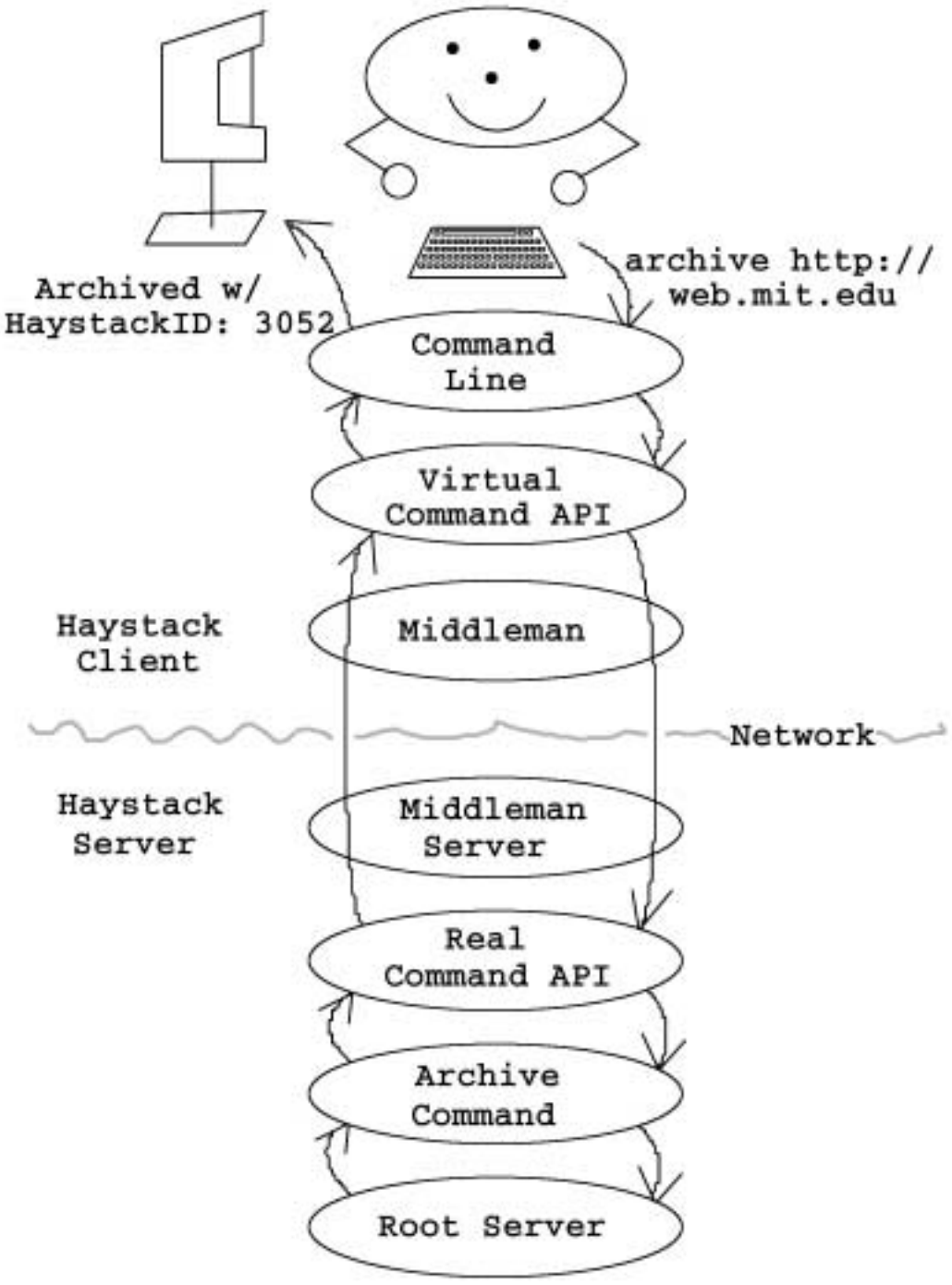

Figure 3-4: A User Requests Archiving of the URL "http://web.mit.edu/".

the same communications and security packages presented in this section.

3.6 Persistent Storage: The Old Way

There were some problems with the old persistent storage mechanism that the triple store will eventually replace. It is difficult to identify the intentions of the old persistence mechanism, since parts of it have been deactivated or are not implemented in accordance with comments and documentation. A brief overview may still prove instructive in work that follows this thesis. It is important to note that the root server still currently uses the old persistent storage tools.

The interface to the old storage module is a persistent hashtable. This hashtable stores *(key, value)* pairs for the Haystack Data Model. Access to persistent hashtables is mediated by a module called the kernel. Transactions would also be mediated by the kernel, but they are currently deactivated in the code.

Many of the ideas of the kernel could be used when the triple store replaces the persistent hashtable if the kernel is fixed. One problem is improper use of variables which can potentially hold transaction handles for use in Haystack threads. All threads share the same transaction handle pointer, and concurrent transactions collide as a result.

One early step in this thesis improves robustness in Haystack with a backup service[23]. This service periodically halts Haystack, copies the entire state of the Haystack Data Model to a backup copy, then allows Haystack to continue. The intent is that if Haystack enters an impersistent state on failure, we can at least revert to a somewhat recent backup copy. While the backup service may help with certain failures, the possibility of reaching an irrecoverable state still exists. If at any point the backup service fires with an impersistent Haystack, the clean backup will be overwritten with a faulty Haystack state, ruining the user's Haystack.

The triple store intends to provide a truly persistent remedy via a working transaction management system that ACID compliant modules can use. If Haystack uses the triple store correctly, it can be a valuable improvement over the root server's old

persistent storage strategy.

In the next chapter we present some of the changes that were necessary to allow the triple store to reuse well written and useful services. The Haystack Trust Model permits reuse of utilities with an architecture very similar to the Haystack Service Model described in this chapter. In many cases design aspects are simply copied over to the reusable model. Through slight modification some services can become reusable and non-root server-dependent utilities, while safely coexisting with the current service architecture.

Chapter 4

The Haystack Trust Model

The Haystack Trust Model is an adaptation of Haystack system design that allows modules to have their own namespaces. One or more instance of a module may be loaded and unloaded as needed. Emphasis is placed on reuse of submodules to avoid wasteful design and implementation overhead whenever possible. It is now much easier to reuse some pieces of the Haystack project.

The Haystack Trust Model (HTM) fixes some of the problems created by the Haystack Service Model, while reusing many earlier Haystack design decisions. The HTM introduces utilities that serve the same purpose as services, but can be used in different modules. The HTM allows multiple instances of utilities to coexist in different modules. By contrast, exactly one instance of a particular service may run at a time, and it must run within the root server module. The HTM also facilitates communication between utilities with semantic descriptions, in much the same way as service intercommunication progresses. Communication is permitted between all utilities in a particular web of trust, as created by a module. New modules should adopt the HTM and include reusable utilities where appropriate.

The creation of the HTM has done little more than make Haystack design more flexible. Most of Haystack remains unchanged. The entire service hierarchy has not been changed, since such a modification would require editing more than one hundred classes. Changing services to the Haystack Trust Model is not absolutely necessary at this point. However, two legacy modules were modified. The security

HsService	: root server	:: NamedUtility	: <arbitrary module>
service	: service name	:: utility	: utility name

Figure 4-1: The Relationship Between Services and Utilities

and communications modules have been adapted to support the more flexible HTM. In order to avoid breaking the root server, these modules currently support both the older Haystack Service Model and the newer Haystack Trust Model.

4.1 A New Distinction: Utilities vs. Services

We present the concept of a `NamedUtility` to improve upon the `HsService` hierarchy. The `NamedUtility` class is the class at the top of the utility hierarchy. Hence all utilities are subclasses of `NamedUtility`. A utility is a generic service that does not require a root server and that permits multiple instances of itself in different modules concurrently. Figure 4-1 demonstrates the relationship between services and utilities. One of the features of a service is the descriptive service name it is given upon instantiation. The service name is used as a description of the service during interservice communication. We use a utility name in place of a service name for utilities in the HTM. Utility names are used in the same way as service names to maintain a constistent design throughout the Haystack project.

A `NamedUtility` is any utility to which we can refer via a utility name description. A `NamedUtility` may optionally register itself with, or *declare itself an available resource to*, a namespace. Other utilities in the same web of trust as that namespace may then access the `NamedUtility` via its utility name. We can register utilities within a single namespace upon instantiation. We can also delay namespace registration. Namespaces for arbitrary modules are presented in more detail in Section 4.3. A utility is also implicitly initializable and closeable. These two methods ensure that the utility itself is dynamically loadable and unloadable.

Some services were modified to run as utilities that may be used in any module. The modification occurred within the communications and security modules. Both

modules are used by the triple store and may be useful to any future module that desires communication and security. These classes are services when instantiated by the root server module, and utilities when instantiated by the triple store or any other module. We still allow them to be services because we require that this thesis leave the root server module unchanged. We allow them to be utilities because they are used by the triple store as well. The confusion remains because the introduction of the HTM could not break the way the root server already functions.

4.2 Haystack Without a Root Server?

Previously, the concept of a Haystack without a root server did not exist. We now present an alternative to that design where the root server merely represents the module that performs all of the IR functionality of Haystack. In our new model we can have arbitrary modules, which may use several utilities, running. A module is any self-contained process running in the Haystack memory space. Modules can talk with different parts of Haystack when permitted by the system. A module might interact with the root server, triple store, or any other module, or it might run completely on its own.

We are now moving towards a new model of the Haystack world, as Figure 4-2 shows. This model shows two different aspects of Haystack. One part has everything that happens inside the Haystack process. The other pieces are processes running outside the Haystack process. The boundary between processes is a grey wavy line in the figure. Arrows indicate communication across the boundaries.

The middle of the figure shows parts of Haystack that can run on the server side. Above and below the grey boundaries are processes which can run in remote programs. In the old model we always had one root server running on the Haystack program server side. Now we may run either a root server or a triple store. Note that a solitary triple store does not support the root server IR capabilities. The design also makes it possible to run one root server and many triple stores concurrently.

The top of Figure 4-2 shows the ways to connect remotely to the root server.

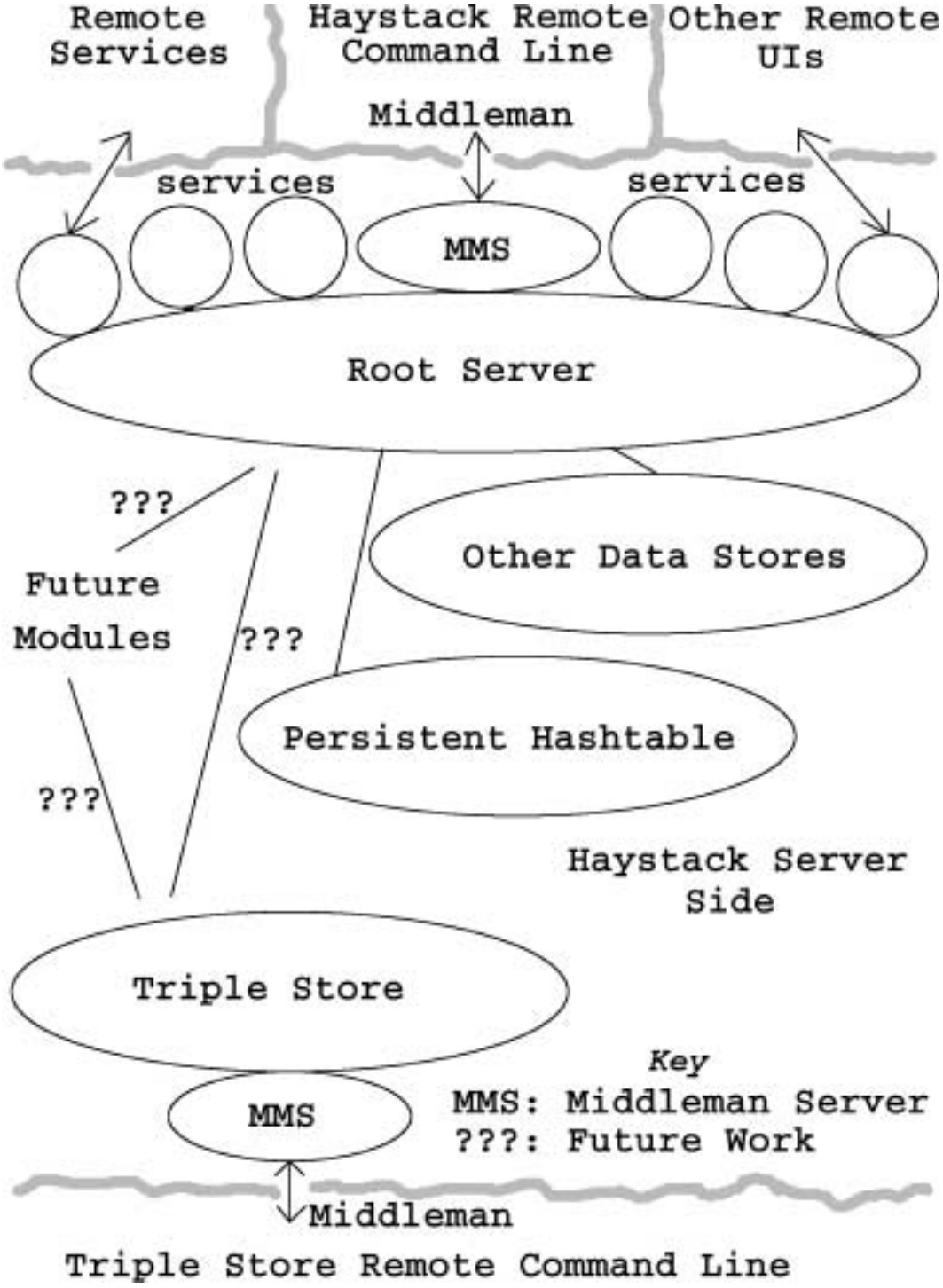

Figure 4-2: The New Haystack Model

There exists the possibility for remote services to operate outside of the root server and communicate with the root server via a special service. There is also a remote command line, which makes use of the communications and security modules as services. The actual dialogue is coordinated by a middleman-to-middleman server connection. The middleman architecture encrypts authentication and the ensuing information exchange using the security services. There are also other remote user interfaces, such as a Web interface, that can connect to appropriate services.

The bottom of the figure shows how one can remotely connect to a triple store. The triple store uses the communications and security modules as utilities. Using them as utilities just means that they use a triple store namespace for interutility communication rather than the root server namespace. Again the actual remote dialogue follows the middleman architecture, including encryption and authentication.

We expect that the root server will eventually migrate to the triple store for persistent storage. Currently the root server uses other data stores such as the persistent hashtable for this purpose. The connection between the triple store and the root server is surrounded by question marks in the figure. The question marks denote possible lines of interactivity in the future. Since the migration to the triple store has not yet begun, we don't know whether we will connect with the triple store directly or via future modules. Note that a future module may be granted permission to interact with the root server, the triple store, or both.

Using the Haystack Trust Model, the triple store is a modular entity whose utilities have less dependency constraints than their service counterparts. We may now run an arbitrary number of triple store modules on their own. We can run a triple store without a root server. The triple store module depends only on the existence of the `haystack.utils` package, the `haystack.security` package, and the `haystack.communications` package. This makes sense since the triple store is a *utility* module that uses *security* for encryption and decryption and *communications* to support remote clients. It should be noted that the `haystack.exceptions` package is actually required as well, because older Haystack modules put their exceptions there. Exceptions are Java objects which are usually used to indicate some critical

or noncritical failure during a method execution.

4.2.1 The Config Utility is Essential (sort of)

Previously, we needed to have a root server and all its core services in order to run any other part of the Haystack program. With the introduction of a standalone triple store, we require only a config utility. When we say the config utility is essential under the HTM, we do not mean that all future modules will need it. We just assume that most modules will want to make use of the configuration capabilities it offers. If there is no desire to use a set of dynamic configurations for a module, it is fine not to have a config utility.

The root server currently runs without a config utility because the `haystack.service` package and all subpackages were never touched during creation of the triple store. The root server runs with its own config service. The config service and config utility support identical functionality except that the service version may only be used within the root server. For the root server to create and use a completely separate module such as the triple store, it must first initialize a global config utility.

4.2.2 Other Essential Utilities

The logging capabilities present in the root server may be modified for future use by utilities. We recognize that logging is probably useful in the regular operation and debugging of any module. For this reason, we may eventually require the existence of a logging utility. It is important that this and any other utility dependencies be examined closely before adding them to the HTM. Otherwise we may end up overconstraining our utility model and creating the same problems we faced with the service model.

4.3 A Per-Module Namespace Utility

The entity a `NamedUtility` registers with is called a name utility. The name utility is a `NamedUtility`, but does not register with itself. The name utility represents a

namespace which is an area of trusted communication between utilities. There is a global namespace and a module-specific namespace. If a `NamedUtility` is registered with the global namespace then anything in the Haystack program is in its trusted web and can communicate with it. If instead the `NamedUtility` registers with a module-specific namespace, access to the utility is limited to those within that namespace.

4.3.1 A Namespace Example

Figure 4-3 depicts the creation of the `JDBCStore` module and its subsequent interaction with the namespace. Module-specific namespaces are created by the command `getNextNameSpaceID` in the `Loadables` class. This returns a key for using the namespace in the future.

We call a module a black box if the internal machinery of the module cannot be exposed. The programmer is restricted to an API of input and output methods. A black box module is created by instantiating and initializing a single instance of a class. We call this instance the controlling class of the module, since the interfaces it implements are the only methods available outside the black box. A module can be a true black box if its controlling class never exposes the namespace key it uses in internal methods. The `JDBCStore` implementation of the triple store is one such module controller.

Time progresses in the figure from top to bottom. To the left of the `JDBCStore` boundary is the module creator's namespace. The inputs and outputs of methods cross this line, but the black box of the `JDBCStore` is maintained so long as no utility pointer or namespace key is sent across the boundary.

4.3.2 Who Controls a Namespace?

The `JDBCStore` creates a namespace when it is instantiated, and hides the namespace key within its private representation. During initialization it can then create new utilities and pass them the namespace key, as with utilities A and B in the figure. Passing the namespace key to each `NamedUtility`'s constructor registers the new

utility in the module's namespace. The utility keeps a copy of the key in order to communicate with other utilities in the namespace.

4.3.3 The Namespace Defines a Web of Trust

The namespace key defines a web of trust for a module. Suppose *methodX()* is invoked, as per Figure 4-3. *Utility A* can use its copy of the key to get the name utility by invoking `Loadables.getNextNameSpaceID()`. *Utility A* can choose to get a pointer to another registered utility using a *get(T)* symantic method call. *Utility A* obtains a pointer to *Utility B*, and interaction follows. Eventually the method returns *o*, some arbitrary object.

The key is verified before the `Loadables` class will return a pointer to the appropriate name utility. Verification is done by checking that it is a pointer to the actual identifier object created when the namespace itself was created. Java prevents other modules from artificially creating this pointer. The only way to grant access to the key is to pass it explicitly to another module. With the key a utility can use semantic descriptions to request communication with other utilities in its namespace. The methods for accessing and managing the namespace are very similar to those in the name service. Methods were kept the same for uniformity. However, the name utility adds the ability to restrict access to its namespace and to create an arbitrary number of namespaces.

There is one fundamental difference between a name utility and the name service. The name service stores information about the services that register with it peristently on disk. The name utility stores nothing perisistently on disk. The reason such capabilities were not included in the name utility is that the triple store module does not require saving the state of any of its utilities. If a new module wants to save this information, the peristent namespace functionality would have to be added.

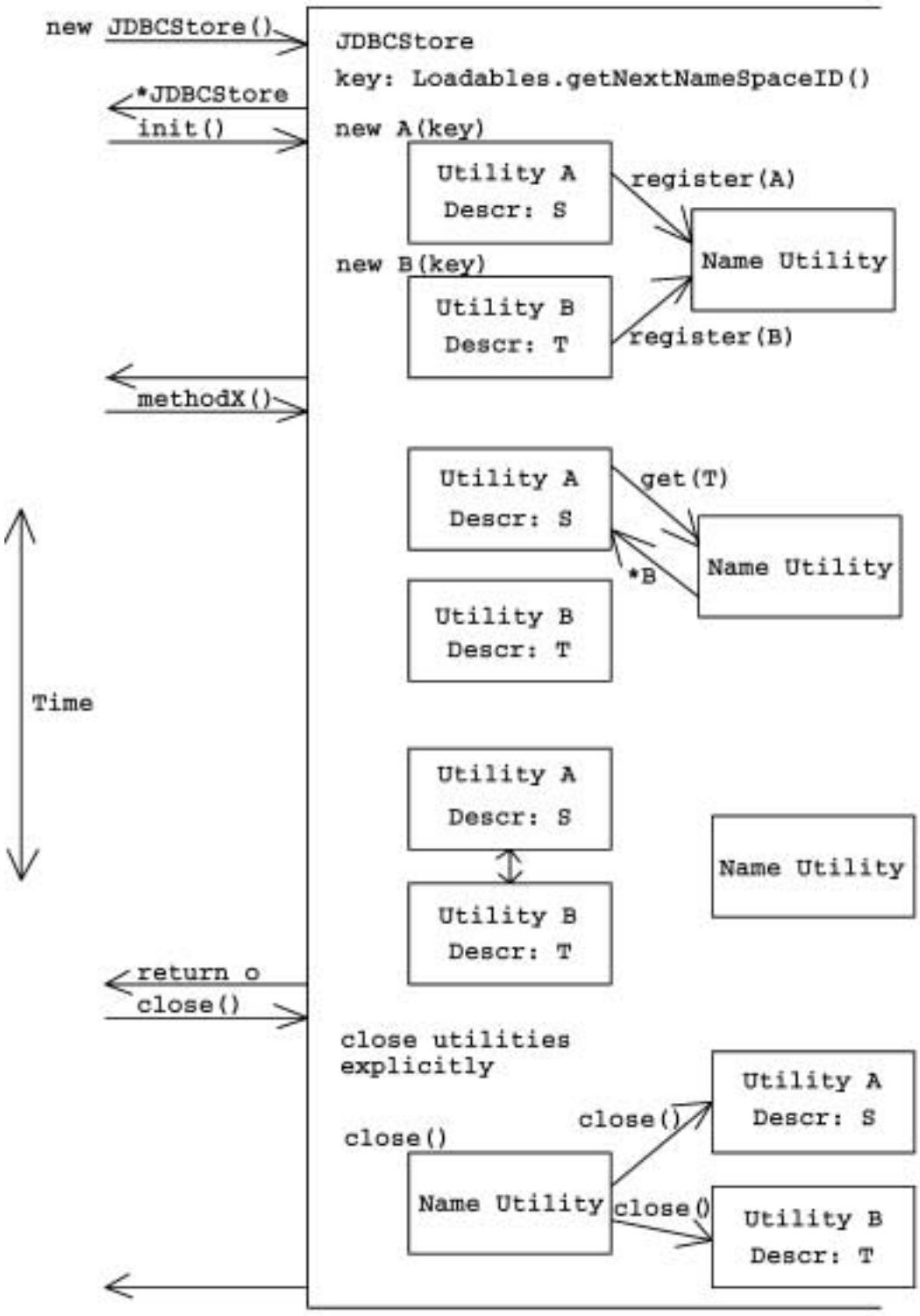

Figure 4-3: A Sample Module Controls its Namespace

4.3.4 Dynamic Inclusion in a Namespace

Another feature of the name utility is that it can dynamically load and unload the entire namespace. When the module controller is created, it creates a namespace. Utilities can be registered in the namespace to interact. For example, a list of utilities stored in the global config utility may contain utilities to load dynamically. If those classes are created with the namespace key argument, they are registered in the namespace.

4.3.5 Cleaning Up a Namespace

To close the module we simply call the *close* method on the module controller as in Figure 4-3. `JDBCStore` in turn asks explicitly loaded utilities to close, and can then ask the namespace itself to close. The name utility iterates through all registered utilities and closes each one. When finished, the namespace, and thus the module itself, has been unloaded.

4.3.6 When is a New Namespace Necessary?

Utilities registered in a particular namespace are unique to that namespace. The utility name description uniquely identifies a single utility. So we know we need a new namespace when we want to create a new instance of some already registered utility, or if we want to create a new web of trust for a submodule. One way to hack around the uniqueness constraint is to modify the utility name with some arbitrary character string extension to one of the identifiers.

Suppose we are within a certain module namespace and want to create a submodule with its own namespace. The advantage of creating a subnamespace is that the black box constraint can be enforced within the submodule. The submodule prevents access to its utilities by keeping its namespace key hidden, and a new web of trust is created.

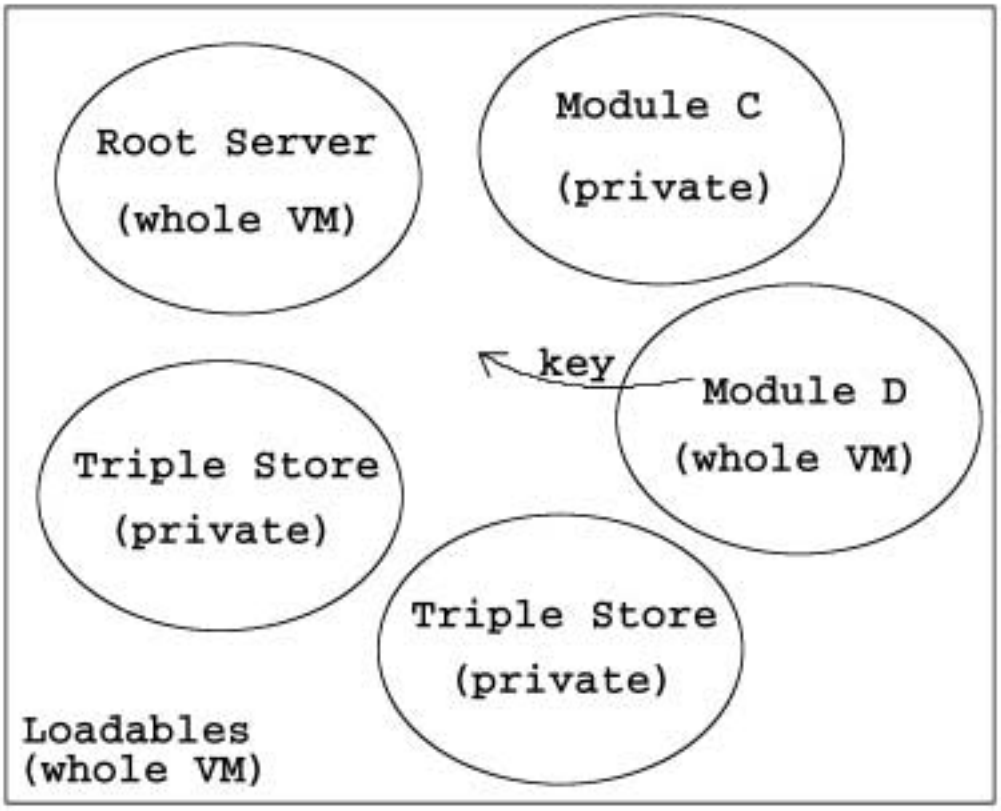

Figure 4-4: The Namespace View of the World

4.4 A More Modular Haystack

We now revisit the new Haystack Trust Model in terms of the namespace structure of all of Haystack. Recall that all new modules may have an associated namespace but that the root server has a special namespace maintained by the name service. Figure 4-4 shows the namespace view of the Haystack architecture. Phrases in parentheses denote the accessability of each module's namespace. Note that anything in the Java Virtual Machine (VM) has access to both the `Loadable`'s global namespace and the root server's name service. *Module D* has returned its namespace key to the rest of the Haystack application. We assume the worst case and say that its namespace is accessible anywhere in the program. *Module C* and the two triple stores have kept their keys and therefore their namespaces private.

Nothing in the root server actually uses the utility loader. We have not yet modified

the service architecture. We could in the future port the root server to the new model and actually make use of a name utility instead of a name service. The benefit would be that more services would be reusable. However, doing so would require modifying every block of code where a service uses the name service. Since this happens very frequently, we may just create a name utility for the root server to use only for new services, avoiding the task of modifying existing services. Right now the change is not critical, as long as we keep in mind that the root server's name service is exposed to all modules within the Haystack application.

4.5 Adding A New Module

The steps for creating a new module are now a lot easier. Suppose we want to introduce the triple store module. To run a triple store we need a controller class such as a `JDBCStore`. `JDBCStore` creates a new namespace in its constructor and initializes any utilities it plans to use, giving the appropriate ones access to the trusted namespace. Explicitly created utilities are core utilities of the module on which other utilities may depend. A middleman server is one utility that the `JDBCStore` can explicitly create. A `JDBCStore` also searches the global config utility for a list of utilities to dynamically load. One utility that the `JDBCStore` dynamically loads is a command API. The command API is given the namespace key on instantiation. Once all utilities have been registered, the namespace is initialized, along with all registered utilities. The `JDBCStore` istance is now ready for use. Closing a `JDBCStore` closes all explicitly created `utilities`, then closes the namespace. Recall that closing a namespace closes all dynamically loaded utilities.

To use the communications module we specify the appropriate middleman server parameters in our `JDBCStore` constructor. `JDBCStore` communications functionality is patterned directly after the way the `root server` uses communications. We give the middleman server the namespace key when calling its constructor. Otherwise the middleman server won't be able to communicate with our module. A specialized command API is optionally loaded according to default configurations. Note that

while we can use a generic middleman server, we must use a module-specific command API implementation that specifies how to deal with commands as they arrive. To do this we simply extend `haystack.utils.CommandAPI` with a new interface, `haystack.ts.CommandAPI`, that can serve as the remote triple store API.

We also have a real and a virtual command API implementation. The real and virtual APIs implement the `haystack.ts.CommandAPI` interface. The `invoke` and `register` methods will do different things depending on whether we are a client (virtual) or server (real) API. The virtual API will simply forward `register` and `invoke` requests to the middleman and wait for the appropriate response. The real API will actually process the `register` or `invoke` request. In the former case a new command is registered in the library of known commands. In the latter case the API checks if there is an appropriate command in its library. If yes, the request is sent to that command. If no, an appropriate error message is returned. Currently only a skeleton for remote command processing has been set up. No commands actually register with the triple store command API.

To run a command line in our `TripleStoreClient`, we extend `haystack.utils.CommandLine` with `haystack.ts.CommandLine`. We keep the generic command line parser, so the only thing we implement is the namespace-key-aware constructor and the `getCommandAPI` method. The former registers the command line in the namespace. The latter selects the correct command API as per the module-specific haystack.ts.CommandAPI's utility name. Note that in order to use a command line we must already have a triple store command API registered in the namespace. This is because command lines do nothing without an API.

Creating a server for a new module is as easy as running a middleman server and creating a command API (and a modified command line if desired). Creating a client is as easy as extending the `haystack.utils.GenericClient` abstract class and implementing methods that assign the appropriate server network address and port, create the appropriate command line, and initialize the appropriate default utilities the client will use. The `TripleStoreClient` is one such extension of the generic client.

This method of addition of a new module is very similar to the steps used by the

root server. The middleman and command architecture are very similar. However, under the new namespace approach we can now have more than one server at a time and reuse things such as the communications module and the generic client.

Chapter 5

The TripleStore Interface

This chapter presents the triple store interface that provides the basic *put* and *get* operations for an RDF model. The triple store is the crux of this thesis. The simplicity and uniformity of this interface will make it a useful data store for the Haystack Data Model. It's transaction contract provides the ability to achieve ACID-compliant robustness. Support for storage of RDF assertions makes it a flexible storage layer. The interface design anticipates multiple triple stores connected to different backends. The interface is a black box module that never exposes anything from its internal web of trust.

As a caveat, remember that simply having a transaction management system does not prevent failure in modules that use them. It is still possible to violate isolation constraints on cached objects, for instance. Refer to Section 2.2.2 for a review of the isolation requirements for an ACID-compliant system. The transaction management system created by the `TripleStore`[1] interface is just a tool box for building an ACID-compliant module. This thesis deals primarily with the `JDBCStore` implementation of the `TripleStore` that is described in Chapter 6. It is possible that some future implementation might fail to adhere to the transaction contract if it is not carefully written.

In this chapter, we present the transaction contract in detail. We then describe

[1] The `TripleStore` is the actual Java interface. It supports the methods of th abstract triple store interface

briefly how a triple store can save a flexible graph for the root server. Next the abstract interface is presented. We then explain how this interface becomes useful as a tool for writing down the meaning of Haystack information in an RDF assertion store. We conclude with a description of the mechanism for communicating with a `TripleStore` both within the local application and remotely via the middleman architecture.

5.1 The Transaction Contract

The triple store provides a transaction management protocol that can be used to achieve robustness in Haystack. The basic contract is simple.

- **BEGIN TRANSACTION:** Begins a new transaction. Transactions may be handled either implicitly within a given thread for a given instance of a triple store, or explicitly via transaction handles passed in to each method.
- **INTERACT WITH TRIPLE STORE:** Once a transaction is open, the *get* and *put* methods of the triple store may be executed. Note that while the actual copies of data in the database will be isolated, this cannot ensure isolation of any cached versions or copies of that data which might be shared with other threads.
- **ABORT TRANSACTION:** This method allows everything executed since the BEGIN TRANSACTION call to be aborted. A transaction may be aborted because some part of the transaction has caused an error, or because the decision has been made to drop a transaction.
- **COMMIT TRANSACTION:** This method allows everything executed since the BEGIN TRANSACTION call to commit and become part of persistent storage. This is the most commonly expected conclusion of a transaction.

In order to commence a transaction, BEGIN TRANSACTION must be executed. Transactions may be maintained implicitly on a per-thread-per-instance basis or explicitly with a transaction handle object. Each implicit transaction handle may be

used in only one thread for a given triple store. An explicit transaction handle may be used to pass a transaction around in several threads. Special attention to isolation and coordination is advised when attempting to use transaction handles in this way. The transaction handle goes through some authentication each time it is used. If at any point an invalid handle is used, or if no transaction is open for the given thread-instance, an appropriate `TransactionException` is thrown. A valid handle is a handle that was obtained from a particular instance of a triple store.

A particular implementation of the triple store may optionally choose to permit a transaction handle from one instance to be used on another instance. Two `JDBCStore` instances allow transaction handles from each other only if they both connect to the same actual underlying database with the same username.

Once a transaction is open, the module using it must realize that it has seized a resource and that it is blocking the rest of the system from using that resource. In the case of the `JDBCStore` this resource is an actual database connection, currently limited in number to a compile-time constant for every instance. Even if a more dynamic connection pool is created in the future, resources will still not be released until either an ABORT TRANSACTION or COMMIT TRANSACTION method is called.

With an open transaction, other triple store methods may be executed on demand. These methods are described in detail in the interface definition discussed in Section 5.3. Note that at any point a `TransactionException` indicates an attempt to use an invalid handle or the absence of a transaction in the given thread and instance. If an `AbortException` is returned during execution of a transaction, there was some problem interacting with the underlying data store. In most cases a problem with the data store indicates the need to abort the transaction.

It is good programming practice to ensure that everything can be executed to completion before the decision is made to commit. When the root server begins to use a triple store for persistence, we must remember that aborts must be possible any time before the commit point.

The COMMIT TRANSACTION and ABORT TRANSACTION methods mark a transaction appropriately, taking care of all disk changes and freeing up the underlying resource so that other parts of the system may use it. Either method may throw an `AbortException`. In the `JDBCStore` implementation an `AbortException` is thrown if the commit or abort throws a SQLException from the Java JDBC layer.

Once the commit or abort method completes, the underlying resource in a transaction is freed. If we are using an implicit transaction handle, then no transaction is open in the thread and instance. If we are using an explicit transaction handle, then it is no longer valid since the resource attached to the handle has been released. We must execute another BEGIN TRANSACTION statement to commence with the next transaction.

There is no need for an explicit recovery procedure for the triple store module. State is meaningless until a commit, and commits are permanent. Our intent is that the triple store can safely fail at any time. Recovery is implicitly handled by the underlying database of the `JDBCStore`.

5.2 A Flexible Graph for Haystack

As mentioned earlier, one of the requirements of our persistent storage interface is that it provide a flexible layer for the Haystack Data Model. The triple store interface permits such a model. Given any graph, we can represent it with ordered triples. Any concept or structure can be described in terms of these triples. The graph may be updated through deletion and insertion of triples and literals.

Once Haystack has migrated to the triple store back end, the objects of the HDM will be stored as triples. Straws, ties, needles, and bales will serialize their states in terms of a series of triples. Saving any of these objects to disk will mean saving all triples representing its state. The contents of needles will be saved by storing or retrieving arbitrary bits. Since there is no update concept in the triple store, there will be no update procedures for Haystack. Updates will be achieved by erasing one piece of information and replacing it with another. This is in keeping with the immutable

CONSTRUCTOR	instantiates a triple store implementation
INIT	initializes the triple store for use
BEGIN TRANSACTION	commences a transaction
COMMIT TRANSACTION	commits an open transaction
ABORT TRANSACTION	aborts an open transaction
PUT ASSERTION	puts a *(subject, object, predicate)* assertion
PUT BITS	puts a collection of bits
GET SUBJECT	gets the subject's URI for an assertion
GET PREDICATE	gets the predicate's URI for an assertion
GET OBJECT	gets the object's URI for an assertion
GET BITS	gets the bits associated with a literal URI
GET LITERAL URI	gets the local URI for a given literal
GET ASSERTIONS BY SUBJECT	gets assertions with specified subject
GET ASSERTIONS BY PREDICATE	gets assertions with specified predicate
GET ASSERTIONS BY OBJECT	gets assertions with specified object
GET ALL ASSERTIONS	gets all assertions
CONTAINS URI	tests whether a URI can be found
EXPUNGE	expunges every occurrence of a URI
CLOSE	closes triple store, cleans up its namespace

Figure 5-1: The Triple Store Abstract Method List

nature of data in the Haystack IR tool. If a particular literal changes state it is no longer the same entity.

5.3 The TripleStore Interface

The TripleStore interface currently supports the abstract methods in Figure 5-1. This list encompasses the abstract functionality of the triple store. There is no type system defined because we expect a client to define its own RDF type system. All resources are simply referred to by their URI when using the triple store methods. There are currently two ways to access the triple store interface: directly through Java methods on a class that implements the `TripleStore` interface or remotely through a set of commands registered in the command API.

The methods INIT and CLOSE are implemented to comply with the guidelines for dynamically loadable and unloadable modules specified by the Haystack Trust Model of Chapter 4. The methods BEGIN TRANSACTION, COMMIT TRANSACTION,

and ABORT TRANSACTION have already been discussed.

We turn now to the methods which may modify the contents on disk. These include PUT ASSERTION, PUT BITS, and EXPUNGE. The PUT ASSERTION command takes an ordered list of three URIs representing resources and creates a triple in the triple store for that assertion. The return type of this method is a URI identifying the assertion resource just created. The PUT BITS command takes an arbitrary set of bits and stores them in the triple store. PUT BITS returns the URI identifying the resource which maps directly to the bit set. Since Haystack is a Java process, a *collection of bits* is an arbitrary object that is stored on disk by serializing the object, or *writing its bits down sequentially.* Returning those bits later corresponds to deserializing the object for use. EXPUNGE performs a search and destroy on the database for a given URI. If the URI refers to a literal, the literal is removed. If the URI refers to an assertion, the assertion is removed. Any assertion that refers to an expunged URI is also expunged. It is recommended, at least for the purposes of an RDF model, that the programmer be sure of what the ramifications are of expunging a URI.

There are also several read-only methods available on the triple store. These include GET SUBJECT, GET PREDICATE, GET OBJECT, GET BITS, GET URI, GET ASSERTIONS BY SUBJECT, GET ASSERTIONS BY PREDICATE, GET ASSERTIONS BY OBJECT, GET ALL ASSERTIONS, and CONTAINS URI. The GET SUBJECT/PREDICATE/PREDICATE method simply returns the URI of the subject/predicate/object of the assertion argument. The GET BITS operation retrieves the bit string for a resource from disk in the form of an object. The GET LITERAL URI method gets the triple store's URI for a given input literal object. The GET ASSERTIONS BY SUBJECT/PREDICATE/OBJECT method gets an enumeration of every assertion with the specified resource as the subject/predicate/object. GET ALL ASSERTIONS will retrieve an enumeration of every assertion in the triple store. CONTAINS URI simply searches the entire triple store for an occurrence of the resource specified. It returns true if and only if the resource is referred to somewhere in the triple store.

We use a URI in place of an assertion, subject, predicate, or object, since all resources can be identified by a URI. We have no type system written into our resource model, and instead refer to a resource by just its URI identifier. A triple store client may choose to store resources of its own type system in the model.

It is important to note that the triple store interface provides no means for an outside entity to create a new URI within its own namespace. The triple store therefore controls its own URI space. The only way a new local URI is created is by adding literals and assertions. By contrast, we may use remote URIs within the triple store's assertions. Everything in the model can therefore be traced back to a literal in our local model or some remote RDF repository.

5.4 A Better Understanding Through RDF Triple Stores

The triple store helps clients record their state uniformly. As we argued in our RDF description of Section 2.3, a client that follows the RDF model can store the meaning of its state with metadata. RDF is powerful enough to allow arbitrary agents that understand the RDF model to translate an arbitrary domain of knowledge into terms it can understand. It does this by looking up the domain ontologies to which the foreign RDF model refers. It can then translate those ontologies into the RDF schema, which the arbitrary agent might then understand. These features may be helpful to the collaborating Haystacks of the future.

5.4.1 A Uniform API for Storing RDF Models

The Haystack IR tool wishes to eventually understand its own data as well as data stored in other Haystacks and foreign RDF applications. Since the triple store is well-suited for storing RDF models, these capabilities will become feasible when Haystack uses the triple store.

Building an RDF onotology to use with a triple store will require some work.

Eventually the Haystack IR tool will store its data model in the triple store according to a Haystack ontology. We can refer the context of our model to the namespace of the RDF schema 1.0, for instance, by storing the URI literals therein. Once we have put the URIs of resources, classes, properties, etc., from that schema into our model, we can begin adding statements to further define the Haystack ontology. We may also wish to refer to the URIs from separate useful RDF ontologies.

In order to add assertions to a triple store we must recognize that the triple store controls its local URI namespace. We can use any local URI that is already stored in the triple store. We also create new local URIs by adding assertions or literals. When literals are stored as bits, a local URI is created and returned to the client for use in assertions. We can also use foreign URIs in our statements. The client can start creating RDF statements out of URIs by using the PUT ASSERTION command.

We introduce in our triple store a small level of indirection where literals are first referred to by a local resource before creating statements with them. This is so we can then refer to the literal with our own local URI identifier. We do this to be consistent with the Haystack Data Model. Every straw has a HaystackID associated with it that will eventually be replaced by a URI. Needles are straws that contain arbitrary bit strings. Since needles will require HaystackIDs, our literals require URIs. The indirection does not break the RDF model and reduces namespace collisions.

We use an monotonically increasing counter to generate the URIs for the triple store. An alternative approach is to use MD-5 hashes to determine URIs. For a literal, the URI would be an MD-5 hash of its bit string. For an assertion, the URI would be an MD-5 hash of some concatenation of the URIs of the subject, predicate, and object. For literals, we would be guaranteed that the same bit string would always resolve to the same URI, and hence be indistinguishable from the other bitwise equivalent bit strings. The chance that any two distinct bit strings would map to the same URI under this strategy is negligible. One disadvantage of this approach is that stating the same assertion again would not add anything to the model, since the same URI would refer to both the original and the new statement. We may want to state the same assertion several times if we are using an evidence accumulation strategy in

our RDF application wherein stating an assertion multiple times might make it more believable.

5.4.2 Reified Statements

In RDF we can write a *statement* either as an ordered sequence of *subject*, *predicate*, and *object* resources, or as a reified statement. A reified statement has the *statement* itself as a resource, with the restriction that upon creation that resource have an *rdf:subject* predicate with *subject* object, an *rdf:predicate* predicate with *predicate* object, and so on. Once we have the reified statement we can attach properties to the *statement* resource to give it more context.

The triple store automatically reifies all statements by returning a URI identifier for statements upon creation. Implicitly present in methods such as GET SUBJECT is an *rdf:subject* predicate with the associated *subject* object attached to the *statement*.

5.5 Using the TripleStore

There are two ways to use the triple store interface's abstract methods. The simple way is directly using Java method calls. The other way is using remote access.

5.5.1 Directly Through Java

To run an instance of a triple store locally we instantiate the `JDBCStore` implementation, and then call its `init` method. We then have a triple store module that we can use according to the abstract methods. The concrete Java forms of these methods, as defined in the `TripleStore` interface, have similar Java method names, arguments, and return types. If we are using an explicit transaction handle, we must use that handle on the correct triple store. If we are using an implicit transaction handle, we must be sure a transaction is open for the current thread and instance. When exceptions are caught we have to decide whether or not to abort the transaction. To close the triple store we should call its `close` method.

5.5.2 Remotely via Middlemen

Remote access is currently accomplished by creating a `JDBCStore` configured to use the middleman communication architecture. The configuration is done by providing the appropriate port number to the `JDBCStore` constructor. The `JDBCStore` then runs a middleman server inside its namespace to wait for client connections. Client connections use the `TripleStoreClient`, a generic client with triple store-specific default settings.

Once a client is running we have an interactive command line to communicate with our triple store. The triple store client can currently provide authentication to log on to the `JDBCStore` remotely. It can also send encrypted string commands to the triple store server. In the future commands received on the server side will perform the appropriate *put* and *get* methods, returning the results to the client.

Understanding how the triple store interface works is important for using the module. In order to improve or fix problems with the module, we must examine the implementation. The next chapter describes some of the high-level components used by the `JDBCStore` module controller to fulfill the tasks required of a triple store.

Chapter 6

Implementating the TripleStore

The `TripleStore` interface is implemented by the `JDBCStore` class. Here we provide some insight into the inner machinery of a `JDBCStore`. We focus on high-level components of the module and discuss design rather than actual Java code. Since the architecture of parts of the module rely on the capabilities of Java programming, particularly the rules about access privileges, a background in Java is helpful in understanding this chapter. The reader may wish to consult a Java reference guide [9].

While this chapter describes only one implementation of the triple store, many variations are possible. Future implementations should adhere to the `TripleStore` interface defined in Chapter 5.

The `JDBCStore` uses the communications implementation described at the conclusion of the previous chapter. The `JDBCStore` can be a server for a remote `TripleStoreClient`. The `TripleStoreClient` sets up a basic command terminal to access a `JDBCStore` directly.

The `JDBCStore` constructor and initialization methods are described in detail. We then discuss the mechanisms by which the triple store transaction contract is upheld. A brief description of URIs follows. We then describe the structure of the database tables that are used in the underlying database. Next a sample *put* and *get* method are shown as representatives of the `JDBCStore` implementation of the triple store abstract methods. We conclude with a note about the way the `JDBCStore` module

shuts down.

6.1 Preparing the JDBCStore Module for Use

Each configuration for a `JDBCStore` instance is either set as the argument of a constructor or, for the default constructor, read from the global config utility. The parameters that may be set include the driver name, database URL, name of the subdatabase to use, the user name to use to log on to the database, and a middleman server port.

The driver name identifies the JDBC driver to use. To use a different vendor's database, simply indicate a different driver to load. This driver must be available to the Java Virtual Machine in its classpath.

The database URL is a unique identifier of the database to which JDBC will connect. A database must be running at that URL.

A subdatabase or data block name indicates the name of the desired database partition to connect to. A subdatabase is appended to the database name to create the illusion that multiple `JDBCStore`s can connect to different databases without requiring administrative configurations for all of them. We internally append the subdatabase name to the database table names and sequence identifiers. Two `JDBCStore`s use the same database but have different subdatabases if they have different data block names and therefore deal with a disjoint set of tables and sequences. The `uriRoot` data member is a concatenation of the database URL and the subdatabase name that uniquely identifies the triple store repository with which we are working.

The user name is used to log in to the database. The password is a secret decrypted from a file via the Haystack security package. The `JDBCStore` connects with the username and Haystack-wide password. The database running at the specified URL must allow the user name and password to login with the appropriate table permissions.

Finally, middleman configurations are used to indicate whether or not we want a particular `JDBCStore` to listen on a port for remote connections. The port may be

either an argument to the constructor or read from the config utility defaults.

With the newly instantiated triple store, we can now initialize it for use. We separate instantiation and initialization to be consistent with the rest of Haystack. The former can be done relatively painlessly while the latter must in some cases be done in a particular order to obey dependencies between utilities.

Initializing the `JDBCStore` first loads in the JDBC driver, and then attempts to create a series of database connections in a connection pool, described in Section 6.2, using the configured URL, user name, and password. Connections are configured to use multi-statement transactions and the TRANSACTION_SERIALIZABLE isolation level. TRANSACTION_SERIALIZABLE isolation provides the strictest level of isolation and helps ensure isolation of data values on disk in a multithreaded application. An internal method is then invoked to test for the existence of the database tables and sequences with which the connections will interact. If they do not already exist, they are created. We continue initialization by explicitly loading in triple store-specific utilities. We then implicitly load the set of classes configured by the config utility defaults. One implicitly loaded utility is the real triple store command API. We use Java reflection to instantiate these classes within our module's namespace by giving them the namespace key. If our module is configured to allow remote connections, we establish a MiddlemanServer connection to listen for remote clients on the appropriate port. Finally, we ask our name utility to initialize itself, which initializes all registered utilities.

6.2 How Transactions Work

The `ConnectionPool` inner class manages database connections for `JDBCStore` transactions. A transaction is opened by binding a connection either to the per instance thread local variable `localCon` or to a `jdbcTransactionHandle` for the client to use. Internally this binding is created when one of the triple store BEGIN TRANSACTION methods is invoked. It either hides the transaction in the instance and thread, or it returns a `jdbcTransactionHandle` to the client. The connection re-

source that has been seized remains seized until a COMMIT or ABORT is executed on the transaction, and then the resource is released from the `localCon` or the associated `jdbcTransactionHandle`. Releasing the connection from a transaction handle instance renders the handle stale or useless.

Internally, the `ConnectionPool` manages the binding and releasing of connections with a list of free resources. The methods GRAB and RELEASE are used to execute these operations, respectively. They obtain a lock on the connection pool object when manipulating the resource lists. The lock prevents situations where updates to the resource lists would collide. Recall the problems with isolation that can occur in the absence of locks.

Once we have established this way of binding actual JDBC connections in our transactions, we can depend on the JDBC driver itself to handle transactions. Transactions are implicitly open on our JDBC connections. The `ConnectionPool`'s GRAB and RELEASE methods ensure that no statements were executed across a given connection from the time of the last COMMIT or ABORT on that connection to the time of the new binding. We just have to make sure that our `JDBCStore` COMMIT and ABORT methods call the connection's COMMIT and ABORT methods, respectively, just prior to releasing the resource. Transactions effectively begin with a BEGIN TRANSACTION call and end with a COMMIT or ABORT call, as desired.

Before any of the basic *put* or *get* operations are attempted on a particular `JDBCStore` instance, the transaction handle is always validated. We validate transaction handles to verify that the underlying database connection exists and can be used to read and write assertions and literals in this instance's namespace. If the method is an implicit transaction method, we just validate the instance's thread local connection. A certain level of security is enforced on explicit transaction handles because the `jdbcTransactionHandle` is a private inner class of the `JDBCStore`, and private inner classes may not be instantiated externally. Since we return transaction handles outside the `JDBCStore` we must protect against the ability to use invalid handles obtained from different instances. We check that the transaction handle is using the same user name to connect to the same database as this `JDBCStore` instance. A

transaction validation error throws a `TransactionException`.

One important feature allows us to use the same `jdbcTransactionHandle` on disjoint triple store repositories if they come from the same database and user name. The `JDBCStore` namespace is a concatenation of the actual database name and the subdatabase. The subdatabase just defines a disjoint set of tables within the same database. All transaction handles from the same database and username have connections with identical capabilities even if they have different subdatabases. We provide the ability to use transaction handles in this case because another Haystack research project needs to use multiple JDBCStores with disjoint database tables [8].

Using transaction handles from different `JDBCStore` instances prevents us from having to write a nested transaction wrapper to perform a series of subtransactions on the `JDBCStore`s. The difficulty with writing a nested transaction is that it must provide the ability to abort all of its containing subtransactions until the nested transaction itself commits [10]. The subtransactions can't commit at exactly the same time, so transaction wrappers must be prepared to undo already committed subtransactions. A nested transaction can only undo a committed subtransaction if it has persistently logged the ability to completely undo the subtransaction. Fortunately we can instead rely on the transactions provided by a database's JDBC driver.

6.3 How URIs Work

`URI`s are used to uniquely identify resources in our `JDBCStore`'s abstract model. Local `URI`s are maintained internally by the `JDBCStore`, which manages its own URI space. The private inner class `jdbcURI` is required to refer to one of a `JDBCStore`'s local URIs. The only way to get a pointer to a `jdbcURI` is to somehow obtain it from a `JDBCStore` method call. `RemoteURI`s may be created anywhere in the Haystack application.

As with transaction handles, we again verify all URIs before allowing a `JDBCStore` *put* or *get* method to continue. We want to ensure that the `JDBCStore` has generated all URIs that refer to its local URI namespace. Violating this constraint could break

our RDF model by allowing clients to attach assertions to a local URI that has not been assigned by the JDBCStore. Instead the JDBCStore manages its own local URI namespace. A jdbcURI is verified to make sure that it has an identical uriRoot. A RemoteURI is verified to make sure that it refers to a resource outside the local URI namespace.

Internally a jdbcURI is just the uriRoot followed by a numeric identifier. Each time a new resource is stored in a triple store, it is given a jdbcURI with unique numeric identifier corresponding to the next value of the sequence in the database. The value of the next available identifier is determined persistently from the database by the JDBCStore private method nextId.

RemoteURIs may be any string that can identify a resource outside of a particular JDBCStore's URI namespace. It has a public constructor that allows clients to instantiate objects referring to remote URIs. Because the JDBCStore manages its own local URI namespace, a RemoteURI may not start with the uriRoot of the JDBCStore in which it is being stored.

6.4 The Database Tables

The JDBCStore implementation uses JDBC connectivity. JDBC supports the SQL standard for table creation and the basic relational database *put* and *get* operations described in Section 2.2.3. Instead of getting into a specific description of how each TripleStore method is implemented we just present the database table structure. These database tables are standard for every JDBCStore instance. We provide a walkthrough of how these tables can be used with a sample JDBCStore *put* and *get* method later in Sections 6.5 and 6.6.

We have omitted the subdatabase suffixes of the table names in our discussion of database tables. Two JDBCStore instances working on the same database with different subdatabases will have different table suffixes.

The database table schema for the JDBCStore can be found in Figure 6-1. The Id_Sequence table is actually just a monochromatically increasing sequence that is

used in the creation of new local URIs for the JDBCStore. The Object_Literals table is used to store the serialization of arbitrary objects to disk. The String_Literals table is used for special storage of Java Strings. The same public methods use the Object_Literals and String_Literals tables depending on if the literal is an arbitrary object or a String. Finally, the RDF_Assertions table stores reified RDF triples.

6.4.1 Storing Literals

The methods that deal with triple store literals use the Object_Literals and String_Literals tables. These methods include PUT BITS, GET BITS, and GET LITERAL URI. We are not guaranteed that two arbitrary Java objects with equivalent state will serialize to the exact same set of bits in our database. To still support storage of arbitrary objects, we keep the Object_Literals table. However we create a special case for character string objects, which we store in the String_Literals table. The way we store character String objects is to convert them to byte strings and compute an MD5 fingerprint of the entire character sequence. We store this fingerprint along with the String literal itself. We can then retrieve a String literal by searching across the MD5 fingerprints. This MD5 does not functionally serve as a URI, and is never exposed to the JDBCStore's client.

Databases differ in the way they handle binary large objects (BLOBS). For example, PostgreSQL used an oid to refer to a BLOB [13]. A database table that stores a BLOB has an oid column that stores a monotically increasing internal identifier for the BLOB. If we store an identical BLOB twice it has a different oid each time. In the Object_Literals table, we can use a bits_id to SELECT the byte stream serialization of the BLOB. However, we cannot use the serialized BLOB to directly SELECT its key after storing it in the table, since the BLOB's byte stream is different from the oid.

The Object_Literals table is comprised of a bits_id and a the_bits column. The bits_id for a new literal is obtained from the Id_Sequence and represents a URI local to the triple store repository. The the_bits column stores a binary large object

Id_Sequence

<count>

Object_Literals

bits_id	the_bits
<uri>	<oid>

String_Literals

string_id	md5	the_literal
<uri>	<md5>	<oid>

RDF_Assertions

assertion_id	subject	predicate	object
<uri>	<uri>	<uri>	<uri>

Figure 6-1: The JDBCStore Database Table Structure

representation of the serialized bits. Since databases differ in the way they handle BLOBS, searching across this column during a GET LITERAL URI command may fail to return the URI for an equivalent already saved input literal object. The PUT BITS and GET BITS operations behave as expected, as objects may be serialized for storage and deserialized for later use.

The `String_Literals` table has a `string_id`, `md5`, and `the_literal` object. As mentioned we use this table for storage and retrieval of String literals. The `string_id` column serves the same purpose as the `bits_id` column of the `Object_Literals` table. It contains a new id obtained from the `Id_Sequence` at insertion time and is a local URI. The `md5` column contains an MD5 hash of the String literal's character sequence. This column is used for searching during a GET LITERAL URI command to obtain the internal URI for a String literal. The PUT BITS and GET BITS operations allow us to store and retrieve the bits for a String literal according to the `the_literal` column. The `the_literal` column saves the String's serialization as a binary large object that can later be deserialized to create an equivalent String literal object.

6.4.2 The RDF Assertions

The JDBCStore's RDF_Assertions table stores the descriptive RDF metadata. Every RDF assertion takes the form shown in Figure 6-1 and is therefore implicitly reified to give the client the ability to attach properties to the statement itself. A local RDF assertion in our JDBCStore may contain any combination of a local or a remote subject, predicate, and object.

The first column is an assertion_id that is a local URI to refer to the statement stored in that database column. Local URIs for new assertions are obtained in the same way as for the bits_id and string_id columns of the literals tables by using the next available id from the Id_Sequence. Each subject, predicate, and object column contains a local or a remote URI that has passed the JDBCStore's URI restrictions.

The bulk of the triple store methods read and write to the RDF_Assertions metadata table. As this table fills up the RDF model in our JDBCStore becomes very complex. We believe that the metadata together with literals will be sufficient to store the Haystack Data Model persistently on disk.

6.5 A Sample Put

The available JDBCStore *put* operations have similar implementations. Here we look at the putAssertion method in the case where a transaction is implicitly open on the instance and thread. This operation takes as input subject, predicate, and object arguments. They correspond to the RDF notions of a *(subject, predicate, object)* triple. The effect of the method is to add a new assertion in our triple store.

Since many similar methods appear in slightly different contexts, we have private helper methods to handle the different possibilities. In this case we call the private method putAssertion with the appropriate JDBC connection object. Since we are using implicit transactions, the connection is just retrieved from the thread local variable localCon. The private method first validates that we have a connection, then validates all of the input URIs. Next we create a unique identifier for the new

assertion by calling `nextId` to obtain an available id from the `Id_Sequence`. We retrieve a prepared statement from the JDBC connection to insert a row into the `RDF_Assertions` table using a SQL INSERT method. We set the `assertion_id` to be the URI within the local URI namespace with the appropriate new numeric identifier. The `subject`, `predicate`, and `object` column values are set to the input URIs. Any combination of local and remote resources may be used in an assertion so long as the URIs have passed verification. Finally we return the `jdbcURI` for the new `assertion_id` to the environment which called the PUT ASSERTION command. If anything goes wrong during the creation and execution of the method, we throw an `AbortException`.

6.6 A Sample Get

As an example *get* operation we will look at the `getSubject` method with an explicit transaction handle. This method takes in the URI of an assertion and returns its subject URI.

First we validate the transaction handle, then we retrieve the connection from the transaction handle. We then call the private helper method `getAssertionCol` with an argument indicating that we are interested in the `SUBJECT` column. This method first validates the connection extracted from the transaction handle and then validates the assertion URI argument. We create a prepared statement to run our database *get* operation with the URI input. We use a SQL SELECT statement to ask for the `subject` of the appropriate row in the `RDF_Assertions` table. This `subject` may either be a local URI or a remote URI. If the assertion is found, the appropriate `jdbcURI` or `RemoteURI` is returned to the client. If anything goes wrong during execution we throw an `AbortException`.

6.7 How The Server or Client Shuts Down

A `JDBCStore` is shut down by calling its `close` method. `close` first closes the connection pool, which iterates through its connections and makes several attempts to close each of them individually. If multiple threads are using a connection and that connection is closed, there is no way to reopen it. A client should not call the `close` method unless it wants to completely shut down the module for all threads. The next step is to close the middleman server if one exists. Finally, the `JDBCStore` closes out its namespace. We return any error messages encountered while closing resources in a `TsCloseException`.

The remote `TripleStoreClient` closes in much the same way. It closes off its middleman and command terminal. Finally, it closes off its namespace. We return error messages in an `HsCloseException`.

Chapter 7

Future Investigation

In this chapter, we present several ideas for future improvements to the triple store interface and `JDBCStore` implementation. We also present ways in which the Haystack Trust Model can be improved.

7.1 Use An Alternative Way to Store Serializable Objects

Two arbitrary objects with equal internal states can serialize to different bits in the underlying database of a `JDBCStore` because binary large objects are not handled the same by all databases. Consider the case where we want to get the URI for an object with state bits *abcdefg*. Under the current serialization model the `JDBCStore` could deserialize every entry in the `Object_Literals` table to compare the deserialized objects to see if they are equal. Instead, we currently compare the serialization bits to the `the_bits` column to find the URI for an object literal. The cost of deserializing the entire `Object_Literals` table from PostgreSQL oids would be prohibitively expensive. It would be better to allow the database to check the table for the bit string *abcdefg* directly without recreating the object. We use an MD5 hash to fingerprint Java String objects for this type of database selection. A similar technique could be introduced to fingerprint other literals so that the `Object_Literals` table may be

searched using MD5 hashes as well.

7.2 Close the Namespace Properly

In discussing the way namespaces in the Haystack Trust Model are closed, we overlook one small detail. When a module asks its name utility to close, and the name utility in turn closes everything registered with it, including itself, it does not actually remove itself from the `Loadables` list of existing namespaces. It would improve modularity if all resources within a module were freed, including the pointer to the name utility itself in the `Loadables` list.

7.3 Add A Local Triple Store Command Line

Currently the `JDBCStore` command line only runs remotely. A simple change would allow the command line to run locally as well. An instance of the exact same command line class could be used for local and remote versions since they should function identically, except that when a local command line asks for its command API it gets a real one while a remote command line gets a virtual one. This change may be as easy as adding an instance of the already existing `haystack.ts.CommandLine` to the `JDBCStore` server.

7.4 Add JDBCStore Caches

There are several places where the `JDBCStore` could be more aware of its memory usage. The bits of a literal could be cached, and for really large bit bins, we could deserialize the object itself and store it in a local cache. Instead we currently deserialize literal objects every time they are needed.

7.5 Fix Broken Connections

The `JDBCStore` implementation currently does not provide any means of repairing a connection if it breaks. This could pose a problem, especially if we keep using connections and breaking them until we have none left to use. The ability to repair JDBC connections to a database backend might be a useful improvement.

7.6 Prevent Hanging Connections

When we try to initialize a JDBC connection to the backend database, the connection hangs if it cannot be created. If we absolutely need a connection to that particular database in order to do anything useful, the current situation may be okay. But there should probably be some sort of timeout period so that connection attempts only hang for a limited time, after which Haystack tries to find some other task to perform, or displays errors to the user on exit.

7.7 Find Out What An Exception on Commit and Abort Means

The meaning of a JDBC SQLException during commit or abort would be useful to know. This meaning could affect the triple store interface transaction contract. For now it may suffice for a client to abort the transaction.

7.8 Discover if Information Sent Along the Wire Via JDBC Connections is Encrypted

It is not clear whether or not communication between JDBC and the backend database server is encrypted. If it is, our system probably has an acceptable security protocol. If it is not, we should explore the possible options to include encryption. The answer

to this question may depend on the implementation of the database vendor's JDBC driver.

7.9 Ensure That the JDBCStore Can Run on Different Backends

While part of the reason for using JDBC was that it permits connections to many different database backends, we have only tested the `JDBCStore` implementation on the PostgreSQL database. There may be statements executed that are database-specific, and will not work when a `JDBCStore` connects to a different database. One anticipated problem area is binary large object storage inconsistencies across databases. It would behove us to test the `JDBCStore` on different databases. Recall that those databases would have to be administratively configured for connectivity with the Haystack username and password, as discussed in Section 2.2.3.

7.10 Find Out Why the PostreSQL Driver Doesn't Appear to Change Transaction Isolation Levels

There is a specific problem observed with the PostgreSQL JDBC driver that we have used to test our `JDBCStore`. According to documentation, PostgreSQL supports the TRANSACTION_SERIALIZABLE JDBC isolation level [27]. However, the numerical value for the isolation level of a JDBC connection to PostgreSQL does not appear to change when we call the appropriate methods to reset the connection's isolation level. One future improvement to would be to figure out why this is so. If we discover that the PostgreSQL driver does not really support this feature, we may have to change database backends.

7.11 Improve Modularity

In order to improve modularity, those services that can be decoupled from the root server perhaps should be. We could then move all non-root server-dependent services into the set of utilities that other modules can use as well. A migration from using the name service to a name utility web of trust might also be beneficial. Switching to a name utility could protect services from being accessed outside the root server module. Fixing any problem with services will likely be very time consuming since there are so many services.

7.12 Make Haystack Use the Triple Store

The root server will eventually use triple stores instead of the persistent storage it uses now. This change will require defining a Haystack RDF ontology to ensure we are writing valid RDF into our triple stores, and then putting the Haystack Data Model objects into triple stores.

7.13 Haystacks Sharing Information

Once Haystack has migrated to a triple store, we can begin thinking about things such as XML serializations of a user's Haystack that another Haystack might understand. Once this has been established, we can start work on the social aspect of sharing information repositories between colleagues.

Chapter 8

Conclusions

We present a dynamic and robust backend solution to the problem of persistence in the Haystack project. This backend, if used soundly, can help ensure that Haystack becomes a reliable data storage tool.

Migrating Haystack to use triple stores will raise many data and application modelling issues. Steps must be taken towards defining a Haystack ontology. That ontology will allow Haystack to better understand the context of its data, simplifying interHaystack collaboration. Haystack can then begin incorporating foreign RDF information repositories into its knowledge base. An expert system will emerge that caters to the user so that a personalized Haystack search becomes a more natural extension of human thought.

There were several lessons learned while designing and implementing the triple store. Any software module that does not have clearly defined requirements is tough to develop. It is also difficult to create a backend module that can be used by an inflexible legacy system. Creating a backend that can run inside the legacy system or on its own also poses a challenge. If the backend uses some relatively new technology, such as RDF, designing a storage module is difficult since few systems already exist that use that technology. Using incorrectly documented third-party systems can also be an arduous task.

We hope that the problems solved and lessons learned by this thesis will improve the Haystack project. The triple store meets the necessary requirements, but only

with future Haystack improvements will it become useful. A Haystack using the triple store interface should be more robust than one using the existing persistence layer. Using the flexible and uniform triple store should make Haystack metadata more easily understandable and therefore more useful.

Appendix A

RDF Formal Grammar

This is the formal RDF grammar. Table A is taken from [26] from Section 6. It is included here for reference. The reader is encouraged to read the descriptions at the source for clarification.

[6.1]	RDF	::= ['<rdf:RDF>'] obj* ['</rdf:RDF>']
[6.2]	obj	::= description | container
[6.3]	description	::= '<rdf:Description' idAboutAttr? bagIdAttr? propAttr* '/>'
		| '<rdf:Description' idAboutAttr? bagIdAttr? propAttr* '>'
		propertyElt* '</rdf:Description>'
		| typedNode
[6.4]	container	::= sequence | bag | alternative
[6.5]	idAboutAttr	::= idAttr | aboutAttr | aboutEachAttr
[6.6]	idAttr	::= ' ID="' IDsymbol '"'
[6.7]	aboutAttr	::= ' about="' URI-reference '"'
[6.8]	aboutEachAttr	::= ' aboutEach="' URI-reference '"'
		| ' aboutEachPrefix="' string '"'
[6.9]	bagIdAttr	::= ' bagID="' IDsymbol '"'
[6.10]	propAttr	::= typeAttr
		| propName '="' string '"' (with embedded quotes escaped)

[6.11] typeAttr ::= ' type="' URI-reference '"'

[6.12] propertyElt ::= '<' propName idAttr? '>' value '</' propName '>'

| '<' propName idAttr? parseLiteral '>'

literal '</' propName '>'

| '<' propName idAttr? parseResource '>'

propertyElt* '</' propName '>'

| '<' propName idRefAttr? bagIdAttr? propAttr* '/>'

[6.13] typedNode ::= '<' typeName idAboutAttr? bagIdAttr? propAttr* '/>'

| '<' typeName idAboutAttr? bagIdAttr? propAttr* '>'

propertyElt* '</' typeName '>'

[6.14] propName ::= Qname

[6.15] typeName ::= Qname

[6.16] idRefAttr ::= idAttr | resourceAttr

[6.17] value ::= obj | string

[6.18] resourceAttr ::= ' resource="' URI-reference '"'

[6.19] Qname ::= [NSprefix ':'] name

[6.20] URI-reference ::= string, interpreted per [URI]

[6.21] IDsymbol ::= (any legal XML name symbol)

[6.22] name ::= (any legal XML name symbol)

[6.23] NSprefix ::= (any legal XML namespace prefix)

[6.24] string ::= (any XML text, with "<", ">", and "&" escaped)

[6.25] sequence ::= '<rdf:Seq' idAttr? '>' member* '</rdf:Seq>'

| '<rdf:Seq' idAttr? memberAttr* '/>'

[6.26] bag ::= '<rdf:Bag' idAttr? '>' member* '</rdf:Bag>'

| '<rdf:Bag' idAttr? memberAttr* '/>'

[6.27] alternative ::= '<rdf:Alt' idAttr? '>' member+ '</rdf:Alt>'

| '<rdf:Alt' idAttr? memberAttr? '/>'

[6.28] member ::= referencedItem | inlineItem

[6.29] referencedItem ::= '<rdf:li' resourceAttr '/>'

[6.30] inlineItem ::= '<rdf:li' '>' value </rdf:li>'

| '<rdf:li' parseLiteral '>' literal </rdf:li>'

		| '<rdf:li' parseResource '>' propertyElt* </rdf:li>'
[6.31]	memberAttr	::= ' rdf:_n="' string '"' (where n is an integer)
[6.32]	parseLiteral	::= ' parseType="Literal"'
[6.33]	parseResource	::= ' parseType="Resource"'
[6.34]	literal	::= (any well-formed XML)

Figure A: Formal Grammar for RDF

Appendix B

RDF Schema

Here is a copy of the XML serialization of the RDF schema [15] that may help in understanding the RDF type system. The namespace URI for the RDF Schema Specification will change in future versions of this specification if the schema changes. This RDF schema includes annotations describing RDF resources defined formally in the RDF Model and Syntax specification, as well as definitions for new resources belonging to the RDF Schema namespace.

```
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<rdfs:Class rdf:ID="Resource">
<rdfs:label xml:lang="en">Resource</rdfs:label>
<rdfs:label xml:lang="fr">Ressource</rdfs:label>
<rdfs:comment>The most general class</rdfs:comment>
</rdfs:Class>
```

Figure B-1: An XML Serialization of the RDF Schema 1.0

```
<rdf:Property about="http://www.w3.org/1999/02/22-rdf-syntax-ns#type">
<rdfs:label xml:lang="en">type</rdfs:label>
<rdfs:label xml:lang="fr">type</rdfs:label>
<rdfs:comment>Indicates membership of a class</rdfs:comment>
<rdfs:range rdf:resource="#Class"/>
</rdf:Property>

<rdf:Property ID="comment">
<rdfs:label xml:lang="en">comment</rdfs:label>
<rdfs:label xml:lang="fr">commentaire</rdfs:label>
<rdfs:domain rdf:resource="#Resource"/>
<rdfs:comment>Use this for descriptions</rdfs:comment>
<rdfs:range rdf:resource="#Literal"/>
</rdf:Property>

<rdf:Property ID="label">
<rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
<rdfs:label xml:lang="en">label</rdfs:label>
<rdfs:label xml:lang="fr">label</rdfs:label>
<rdfs:domain rdf:resource="#Resource"/>
<rdfs:comment>Provides a human-readable version of a resource
name.</rdfs:comment>
<rdfs:range rdf:resource="#Literal"/>
</rdf:Property>

<rdfs:Class rdf:ID="Class">
<rdfs:label xml:lang="en">Class</rdfs:label>
<rdfs:label xml:lang="fr">Classe</rdfs:label>
<rdfs:comment>The concept of Class</rdfs:comment>
<rdfs:subClassOf rdf:resource="#Resource"/>
</rdfs:Class>

<rdf:Property ID="subClassOf">
<rdfs:label xml:lang="en">subClassOf</rdfs:label>
<rdfs:label xml:lang="fr">sousClasseDe</rdfs:label>
<rdfs:comment>Indicates membership of a class</rdfs:comment>
<rdfs:range rdf:resource="#Class"/>
<rdfs:domain rdf:resource="#Class"/>
</rdf:Property>
```

Figure B-1: An XML Serialization of the RDF Schema 1.0

```
<rdf:Property ID="subPropertyOf">
<rdfs:label xml:lang="en">subPropertyOf</rdfs:label>
<rdfs:label xml:lang="fr">sousPropritDe</rdfs:label>
<rdfs:comment>Indicates specialization of properties</rdfs:comment>
<rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-
ns#Property"/>
<rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-
ns#Property"/>
</rdf:Property>

<rdf:Property ID="seeAlso">
<rdfs:label xml:lang="en">seeAlso</rdfs:label>
<rdfs:label xml:lang="fr">voirAussi</rdfs:label>
<rdfs:comment>Indicates a resource that provides information about the subject
resource.</rdfs:comment>
<rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
<rdfs:domain rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
</rdf:Property>

<rdf:Property ID="isDefinedBy">
<rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
<rdfs:subPropertyOf rdf:resource="#seeAlso"/>
<rdfs:label xml:lang="en">isDefinedBy</rdfs:label>
<rdfs:label xml:lang="fr">esDfiniPar</rdfs:label>
<rdfs:comment>Indicates a resource containing and defining the subject
resource.</rdfs:comment>
<rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
<rdfs:domain rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
</rdf:Property>

<rdfs:Class rdf:ID="ConstraintResource">
<rdfs:label xml:lang="en">ConstraintResource</rdfs:label>
<rdfs:label xml:lang="fr">RessourceContrainte</rdfs:label>
<rdf:type resource="#Class"/>
<rdfs:subClassOf rdf:resource="#Resource"/>
<rdfs:comment>Resources used to express RDF Schema
constraints.</rdfs:comment>
</rdfs:Class>
```

Figure B-1: An XML Serialization of the RDF Schema 1.0

```
<rdfs:Class rdf:ID="ConstraintProperty">
<rdfs:label xml:lang="en">ConstraintProperty</rdfs:label>
<rdfs:label xml:lang="fr">ProppritContrainte</rdfs:label>
<rdfs:subClassOf rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-
ns#Property"/>
<rdfs:subClassOf rdf:resource="#ConstraintResource"/>
<rdfs:comment>Properties used to express RDF Schema
constraints.</rdfs:comment>
</rdfs:Class>

<rdfs:ConstraintProperty rdf:ID="domain">
<rdfs:label xml:lang="en">domain</rdfs:label>
<rdfs:label xml:lang="fr">domaine</rdfs:label>
<rdfs:comment>This is how we associate a class with
properties that its instances can have</rdfs:comment>
</rdfs:ConstraintProperty>

<rdfs:ConstraintProperty rdf:ID="range">
<rdfs:label xml:lang="en">range</rdfs:label>
<rdfs:label xml:lang="fr">tendue</rdfs:label>
<rdfs:comment>Properties that can be used in a
schema to provide constraints</rdfs:comment>
<rdfs:range rdf:resource="#Class"/>
<rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-
ns#Property"/>
</rdfs:ConstraintProperty>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property">
<rdfs:label xml:lang="en">Property</rdfs:label>
<rdfs:label xml:lang="fr">Proprit</rdfs:label>
<rdfs:comment>The concept of a property.</rdfs:comment>
<rdfs:subClassOf rdf:resource="#Resource"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Literal">
<rdfs:label xml:lang="en">Literal</rdfs:label>
<rdfs:label xml:lang="fr">Littral</rdfs:label>
<rdf:type resource="#Class"/>
<rdfs:comment>This represents the set of atomic values, eg. textual
strings.</rdfs:comment>
</rdfs:Class>
```

Figure B-1: An XML Serialization of the RDF Schema 1.0

```
<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-
ns#Statement">
<rdfs:label xml:lang="en">Statement</rdfs:label>
<rdfs:label xml:lang="fr">Dclaration</rdfs:label>
<rdfs:subClassOf rdf:resource="#Resource"/>
<rdfs:comment>This represents the set of reified statements.</rdfs:comment>
</rdfs:Class>

<rdf:Property about="http://www.w3.org/1999/02/22-rdf-syntax-ns#subject">
<rdfs:label xml:lang="en">subject</rdfs:label>
<rdfs:label xml:lang="fr">sujet</rdfs:label>
<rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-
ns#Statement"/>
<rdfs:range rdf:resource="#Resource"/>
</rdf:Property>

<rdf:Property about="http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate">
<rdfs:label xml:lang="en">predicate</rdfs:label>
<rdfs:label xml:lang="fr">prdicat</rdfs:label>
<rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
<rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-
ns#Statement"/>
<rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-
ns#Property"/>
</rdf:Property>

<rdf:Property about="http://www.w3.org/1999/02/22-rdf-syntax-ns#object">
<rdfs:label xml:lang="en">object</rdfs:label>
<rdfs:label xml:lang="fr">objet</rdfs:label>
<rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-
ns#Statement"/>
</rdf:Property>

<rdfs:Class rdf:ID="Container">
<rdfs:label xml:lang="en">Container</rdfs:label>
<rdfs:label xml:lang="fr">Enveloppe</rdfs:label>
<rdfs:subClassOf rdf:resource="#Resource"/>
<rdfs:comment>This represents the set Containers.</rdfs:comment>
</rdfs:Class>
```

Figure B-1: An XML Serialization of the RDF Schema 1.0

```
<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag">
<rdfs:label xml:lang="en">Bag</rdfs:label>
<rdfs:label xml:lang="fr">Ensemble</rdfs:label>
<rdfs:subClassOf rdf:resource="#Container"/>
</rdfs:Class>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq">
<rdfs:label xml:lang="en">Sequence</rdfs:label>
<rdfs:label xml:lang="fr">Squence</rdfs:label>
<rdfs:subClassOf rdf:resource="#Container"/>
</rdfs:Class>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Alt">
<rdfs:label xml:lang="en">Alt</rdfs:label>
<rdfs:label xml:lang="fr">Choix</rdfs:label>
<rdfs:subClassOf rdf:resource="#Container"/>
</rdfs:Class>

<rdfs:Class rdf:ID="ContainerMembershipProperty">
<rdfs:label xml:lang="en">ContainerMembershipProperty</rdfs:label>
<rdfs:subClassOf          rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-
ns#Property"/>
</rdfs:Class>

<rdf:Property rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#value">
<rdfs:label xml:lang="en">object</rdfs:label>
<rdfs:label xml:lang="fr">value</rdfs:label>
</rdf:Property>

</rdf:RDF>
```

Figure B-1: An XML Serialization of the RDF Schema 1.0

Appendix C

Glossary of Haystack Terminology

- **AbortException**: Part of the triple store package, an AbortException indicates some problem executing a method on the underlying database as part of a transaction. The usual course of action will be for the triple store to abort the transaction.
- **archive command**: The command that stores new information into a user's Haystack. Archiving is the processes of recording new documents or other resources for later retrieval.
- **backup service**: A service that takes a periodic snapshot of a user's Haystack. The intent is that a user can revert to an old snapshot if his or her Haystack becomes corrupted.
- **bale**: A bale is a special type of straw that represents a collection of straws. We can add metadata to the entire collection by attaching straws to a bale.
- **command API**: The interface for registering and invoking commands for a module. The command API architecture is used in both the root server and the triple store modules.
- **command line**: A text-based user interface that parses inputs from a terminal and outputs results therein. Command lines exist for both the root server and triple store modules.

- **config service**: The service that manages configurations for use in the root server.

- **config utility**: The utility that manages global configurations that are intended to serve as defaults for instantiation of new modules. The config utility is currently identical to the config service except that it can run outside the root server. A global config utility is currently essential before creating and running a triple store.

- **core services**: Those services in the root server that other services may depend on. The assumption is that every service can use any of the core services.

- **Haystack Data Model (HDM)**: The fundamental data objects used by the Haystack IR tool.

- **Haystack Service Model (HSM)**: The model of dynamically loadable services running around within Haystack to perform different tasks.

- **Haystack Trust Model (HTM)**: An enhancement to the Haystack Service Model that introduces the concept of utilities. Focus within the HTM is on maintaining modularity and reusability of code.

- **HaystackID**: An identifier that refers to any straw in the Haystack Data Model.

- **HsService**: The Java class at the root of the service hierarchy. An HsService is a service that can be referred to via a semantic service name description.

- **JDBCStore**: The Java implementation of the triple store module. A JDBCStore connects to a database via the Java DataBase Connectivity API.

- **kernel**: A module which limits access to critical data within the Haystack IR tool. Examples of critical data are Haystack Data Model elements and transaction handles for modification of disk data. Transactions are currently turned off in the kernel. A modification of the kernel may be used in the future

to restrict access to the triple store and its transaction handles. The root server currently protects important data and pointers using this module.

- **Middleman**: The client for one form of remote communication in Haystack. It uses security objects to authenticate and encrypt a dialogue with a MiddlemanServer. A Middleman can be either a service if created by the root server or a utility if created by the triple store.

- **MiddlemanServer**: The server for one form of remote communication in Haystack. It uses security objects to authenticate and encrypt dialogues with any Middlemen that connect to it. A MiddlemanServer can be either a service if created by the root server or a utility if created by the triple store.

- **name service**: The service that other services use to initialize interservice communication. The name service can be used to dynamically register and load services. It allows services to talk to each other via semantic service name descriptions. The name service also stores information about the services loaded into it persistently on disk. The name service is the root server's namespace. It may be accessed from anywhere in the Haystack process.

- **name utility**: The utility that other utilities use to initialize interutility communication. The name utility can be used to dynamically register and load utilities. A name utility is a web of trust that privileged utilities can use to talk to each other via semantic utility name descriptions. The name utility is an arbitrary module's namespace. The global name utility may be accessed from anywhere in the Haystack process, but a module-specific name utility can only be accessed by privileged utilities.

- **NamedUtility**: The Java class at the root of the utility hierarchy. A named utility is a utility that can be referred to via a semantic utility name description.

- **needle**: A needle is a special type of straw that contains unprocessed data or collections of bits. The text of a document could constitute one kind of needle.

- **persistent hashtable**: The existing persistent interface to disk that the root server uses to store its data model. A persistent hashtable stores (key, value) pairs.

- **query service**: A service that processes queries and attempts to generate suitable result sets.

- **real**: Belonging to the server side. Typically real services or utilities perform the actual work that virtual counterparts request.

- **root server**: The Information Retrieval module of the Haystack project. Eventually we anticipate that the root server will use the triple store as a persistent interface to store its data model.

- **service**: An entity that runs in the root server to aid in some Information Retrieval purpose. Services can run only in the root server. Their instances are restricted by uniqueness within a Haystack process.

- **service name**: The semantic description of a service. Currently comprised only of character strings, we anticiapte other descriptions in the future.

- **straw**: A straw is the root of the Haystack Data Model hierarchy. All data model objects are straws. A straw has a unique HaystackID that refers to it.

- **tie**: A tie is a special type of straw that forms a labelled, directed edge from one straw to another. Ties are the basic relational entities in the Haystack Data Model and together with the objects they connect constitute an assertion.

- **TransactionException**: An exception which indicates an attempt to use an invalid handle or the absence of a transaction for the specified thread and instance of a triple store.

- **TransactionHandle**: The Java interface that refers to a transaction handle to a triple store. A transaction handle may be used to make a series of actions on the triple store appear as though they were one atomic action.

- **triple store**: A module which stores information on disk in the form of ordered triples while offering a transaction contract that can provide persistence if properly used. Each triple is called an assertion.

- **TripleStore**: The Java interface to the triple store module.

- **TripleStoreClient**: The Java implementation of the remote triple store client. Currently the remote client can just connect and run simple commands on the server side.

- **URI**: The Java interface that refers to a resource in the triple store. We use URIs to refer to resources in the triple store in much the same way that HaystackIDs are used in the Haystack Data Model.

- **utility**: An entity which can have many instances running in different modules concurrently. Compare to the notion of a service.

- **utility name**: A semantic description of a utility. Currently comprised only of character strings, we anticipate other descriptions in the future. Similar to a service name except created for use in a name utility.

- **virtual**: Belonging to the client side. Typically virtual services or utilities will have a real counterpart to whom the actual work is relayed.

Bibliography

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*, section 3.2, pages 236–251. The MIT Press with The McGraw-Hill Companies, Inc., Cambridge, MA, second edition, 1996.

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

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

[4] Krishna Bharat. Searchpad: Explicit capture of search context to support web search. http://www9.org/w9cdrom/173/173.html.

[5] Bert Boss. Xml in 10 points. Technical report, W3C, http://www.w3.org/XML/1999/XML-in-10-points, March 1999.

[6] J. Budzik and K. J. Hammond. User interactions with everyday applications as context for just-in-time information access. *Proceedings of the 2000 International Conference on Intelligent User Interfaces*, 2000.

[7] Oracle Corporation. Oracle8 server sql reference, release 8.0. http://www.oradoc.com/ora8doc/DOC/server803/A54647_01/toc.htm.

[8] Dominic D'Aleo. A versioning layer for haystack. Master's thesis proposal, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 2001.

[9] David Flanagan. *Java in a Nutshell*. The Java Series. O'Reilly & Associates, Inc., Sebastopol, CA, third edition, November 1999.

[10] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management. Morgan Kaufman Publishers, Inc., San Francisco, CA, 1993.

[11] Philip Greenspun. Sql for web nerds. http://photo.net/sql/.

[12] Philip Greenspun. *Philip and Alex's Guide to Web Publishing*, chapter 12, pages 325–364. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1999.

[13] The PostgreSQL Global Development Group. Postgresql. An open-source project with links to mirrors found at http://www.postgresql.org/ and a U.S. WWW mirror at http://postgresql.readysetnet.com/.

[14] W3C Consortioum XML Working Group, Special Interest Group, and W3C Metadata Activity. Namespaces in xml. Technical report, W3C, http://www.w3.org/TR/1999/REC-xml-names-19990114, January 1999.

[15] W3C RDF Schema Working Group. Resource description framework (rdf) schema specification 1.0. Technical report, W3C, http://www.w3.org/TR/2000/CR-rdf-schema-20000327, March 2000.

[16] W3C XML Core Working Group. Extensible markup language (xml) 1.0 (second edition). Technical report, W3C, http://www.w3.org/TR/2000/REC-xml-20001006, October 2000.

[17] Marti A. Hearst. Next generation web search: Setting our sites. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2000. http://www.research.microsoft.com/research/db/debull/A00sept/hearst.ps.

[18] INT Media Group Incorporated. Ftp - webopedia definition and links. http://www.pcwebopedia.com/TERM/F/FTP.html.

[19] Daniel D. Lee and Sebastian Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401:788–791, October 1999.

[20] Christopher Locke. Accelerating toward a better search engine. *Red Herring*, March 2001. http://www.herring.com/index.asp?layout=story& channel=10000001&doc_id=1450018145.

[21] Aidan Low. A folder-based implementation of a graphical interface to an information retrieval system. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1999.

[22] et al. Lynn Andrea Stein. Annotated daml ontology markup. Technical report, DAML, http://www.daml.org/2000/10/daml-walkthru, October 2000.

[23] Kenneth D. McCracken. Dynamic and robust data storage and retrieval in the haystack system. Master's thesis proposal, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 2000.

[24] M. J. McGill and G. Salton. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, NY, 1983.

[25] Sun Microsystems. Jdbc data access api. http://java.sun.com/products/jdbc/index.html.

[26] W3C RDF Model and Syntax Working Group. Resource description framework (rdf) model and syntax specification. Technical report, W3C, http://www.w3.org/TR/1999/REC-rdf-syntax-19990222, February 1999.

[27] Peter T Mount. Postgresql jdbc driver. An open-source project at http://jdbc.postgresql.org/.

[28] MIT Department of Electrical Engineering and Computer Science. 6.033 draft of chapter eight: Atomicity, April 1999. M. Frans Kaashoek taught the course during the term these notes were distributed.

[29] George Reese. *Database Programming with JDBC(TM) and Java(TM).* The Java(TM) Series. O-Reilly and Associates, Inc., Sebastopol, CA, second edition, August 2000. JDBC and Java are trademarks of Sun Microsystems, Inc.

[30] Michael Stonebraker. Object-relational dbms - the next wave. http://db.cs.berkeley.edu/papers/Informix/www.informix.com/informix/corpinfo/zines/whitpprs/illuswp/wave.htm.

[31] Danny Sullivan. Being search boxed to death. *The Search Engine Report*, March 2001. http://searchenginewatch.com/sereport/01/03-boxed.html.

[32] Inc. Sun Microsystems. Java 2 platform, standard edition. http://java.sun.com/products/jdk/1.2/docs/api/java/sql/package-summary.html.

[33] Inc. Sun Microsystems. The java tutorial. http://java.sun.com/docs/books/tutorial/.

[34] Jaime B. Teevan. Improving information retrieval with textual analysis: Bayesian models and beyond. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 2001.

[35] Dallan Quass William Cohen, Andrew McCallum. Learning to understand the web. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2000. http://www.research.microsoft.com/resarch/db/debull/A00sept/cohen.ps.