Versioning for the Haystack System

by

Dominic J D'Aleo

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

June 2002

© Massachusetts Institute of Technology, MMII. All rights reserved.

Author ..............................................................

Department of Electrical Engineering and Computer Science

June 3, 2002

Certified by ..........................................................

David Karger

Associate Professor

Thesis Supervisor

Certified by ..........................................................

Lynn Stein

Associate Professor

Thesis Supervisor

Accepted by ..........................................................

Arthur C. Smith

Chairman, Department Committee on Graduate Students
Versioning for the Haystack System

by

Dominic J D’Aleo

Submitted to the Department of Electrical Engineering and Computer Science
on June 3, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, the design and implementation of a repository for electronic information is presented. While designed with the Haystack system in mind, the repository can easily be used by anyone familiar with RDF. This repository performs the basic task of storing the information that the user collects, in addition to automatically performing several other tasks in an effort to make retrieving the information simple and efficient. The additional tasks involve the storing of when and why information enters the repository. The hope is that this additional data will help the user when searching their repository. A person’s sense of time and data dependence is a strong organizing principle that can help them locate a file simply because they remember when or why they archived it, or perhaps when they used it last. This repository allows a user to narrow the search of their information space in regards to time and dependence.

Thesis Supervisor: David Karger
Title: Associate Professor

Thesis Supervisor: Lynn Stein
Title: Associate Professor
Acknowledgments

First and foremost, I would like to thank my family. They have provided the foundation and support throughout my entire life that has allowed me to reach this point, and for that I am forever grateful. I would like to thank my advisors, David and Lynn, for their advice, patience and support throughout this entire process. I learned more than I ever hoped for while working on the Haystack project, and to them and the other people involved with Haystack I owe a great debt of gratitude. Finally, I would like to thank Lindsay. In her own way she has inspired me to work hard every day to achieve my full potential, and hopefully this is the first step along that path. Without her I fear I may have never finished this thesis, and for that I perhaps owe her the greatest thanks.
3.2.2 Crystal Ball ........................................... 38
3.2.3 Rewind ........................................... 39
3.3 Dependency ........................................... 39
  3.3.1 WhoCreated ...................................... 40
  3.3.2 WhyCreated ...................................... 41
  3.3.3 Time Reliance .................................... 41
3.4 Deletion ........................................... 42

4 Interface ............................................. 49
  4.1 Ken's interface note .................................. 49
  4.2 Getting Started with the fancy triple store ............ 49
  4.3 Using the fancy triple store ......................... 50
  4.4 The Evolution of the Interface ....................... 52

5 Implementation ....................................... 55
  5.1 Data-dependency Layer ............................... 55
  5.2 Time Layer ......................................... 56
  5.3 Transactions ....................................... 57
  5.4 Put operations ..................................... 58
  5.5 Get operations ..................................... 59
  5.6 Deletion and Expunge ................................ 61
  5.7 Undeletion ......................................... 63
  5.8 The Evolution of the Implementation ................ 64

6 Conclusions and Future Work .......................... 65
  6.1 Future Work ....................................... 65
  6.2 Conclusions ....................................... 66

A Plain Triple Store Interface ......................... 69
List of Figures

1-1 A time versioning tree for a document ................. 16
1-2 A view of data dependency versioning ................. 17

2-1 An RDF triple - *statement* ......................... 20
2-2 Reification of an RDF triple - *statement* .......... 22
2-3 The Haystack data model ............................ 24
2-4 The Star Graph for HsTypeGuesser ................. 31
2-5 A BABYL document in the HDM ...................... 32

3-1 A search interval and the periods that a piece of data exists ...... 45
3-2 A view of several pieces of data and the dependencies between them . 47
Chapter 1

Introduction

1.1 Haystack

The work that was done in this thesis is intended for use with the Haystack system [7]. While eventually the design evolved to the point where the data repository could be used by any application familiar with its interface, it was designed with Haystack in mind.

Haystack is also an information repository, but much more sophisticated than the one that is discussed in this thesis. Like this one, Haystack’s main goal is to provide a useful way for computer users to store and access their information. Haystack goes a few steps further however. Haystack’s main goal is to personalize itself to a user. By learning about a user of the system when they use it, Haystack hopes to suit itself to that user. I discuss in more detail all the goals of Haystack in the next section.

1.2 Haystack Project Aims

As further background into the coming about of what was done in this thesis, I present some of the aims of the Haystack project. As previously stated, the Haystack Project is intended to create an information repository that maintains a consistent and persistent state, while trying to personalize itself to a user.

Through the use of transactions, Haystack maintains a consistent and persistent
state. The foundation of querying in Haystack is through the use of natural text searches, analogous to those performed on popular Web sites such as Yahoo!. By implementing algorithms from the field of artificial intelligence, Haystack attempts to use feedback from the user to learn about their needs. When the user gets a result set from a query, they have the option of stating that a particular result was quite relevant, or perhaps not relevant at all. Haystack then uses this information to adapt to the user and return better results in the future. A goal of the Haystack project is improve upon the AI that is being used, and adjust to a user as best possible [13].

As important as allowing the user to search their information is allowing the user to annotate their information to make the search tool much more powerful. The Haystack project aims to allow a user to add metadata to information already in their repository, hopefully information that the system might have a difficult time deducing otherwise, and use that information to make search better. One of the goals of this thesis is to deduce more information that would be useful to the user, thus removing some of the burden of manually adding metadata.

1.3 The Problem

The initial goal that resulted in the work presented in this thesis was to solve a problem we were having with Haystack. The problem was that when a user archived a document that was already in their Haystack, Haystack would not realize it. We decided to take an algorithms approach. The problem itself was simple, and could have been solved in a simple manner by doing a bit by bit comparison of a new document to those documents already in the Haystack. This would prevent the main aspect of the problem, which is that there would be two or more identical copies of the same document. An algorithms approach, however, seemed interesting because it would solve the problem in a much more thorough manner and add a feature that might be attractive to a user. The idea was to apply known versioning algorithms, or perhaps an algorithm that could be derived from those. A versioning algorithm would keep track of all the documents in a Haystack in such a way that if a new
or identical version of a document already in the Haystack was archived, it would recognize it as such and make a note that it was related to a previously archived document in that way. This was superior to making a direct bit by bit comparison of a new document to the documents in the Haystack because it added the goal of recognizing a document as a new slightly changed version of a previous document. This concept came to be called “content versioning”, and I will refer to it as such in this thesis.

1.4 Versioning

Before discussing content versioning, it is important to know that the final design of versioning in this thesis is in two forms, neither of which is content versioning. The first is “time versioning”. This is the tracking of change in the Haystack in terms of when the change occurs. When a document enters the Haystack, the time that it does is noted and stored as a separate piece of information. The other is “data dependency versioning”. This is the tracking of change in the Haystack in terms of why it occurs. When a document enters the Haystack, the entity placing it there is noted and stored as a separate piece of information. The entity changed the Haystack, and thus is considered the answer to ‘Why did the Haystack change?’.

We now describe how the idea of content versioning, and how the approach to solving it led to the ideas of time and data dependency versioning. After mulling over the idea of content versioning, and how to tackle such a problem, it became apparent that time was entangled with the idea. Knowing the relative age of documents in a Haystack to each other would be a useful tool in determining if a new document was just a new version of an already existing one.

If it is known that a document entering the system was composed before other documents already in the system even existed, one can deduce that it is not a new version of any of those documents. It is important to note that it was assumed that the time information (time information refers to the real world time a document was created) relative to a document would be extracted from the document itself. In the
case of an email, which contains the time in the body of the document, it would be simple to just parse the email and extract the time.

Having time information would also be useful once versioning lines became versioning trees. (see Figure 1-1) In Figure 1-1, one can see how a user was working on and creating new versions of a document. The began with version 1, and progressed through revisions until they reached version 4. At that point, they decided that were unhappy with their recent revisions, and decided to go back and work on version 2 again. This resulted in version 5, and eventually version 6. In this way, versioning does not necessary only occur along straight lines. If versioning only occurred along lines, it would be easy to determine which documents arose first. That information could be useful to a user who is searching for a particular version of a document in their Haystack. An example of this would be if a user is looking for an old version of a paper they are writing because they don’t like the direction they have taken with it and they want to go back a little and start over. Once versioning lines become versioning trees though, it might be harder to remember which documents entered the repository first and the user might have to spend a lot of time looking at all the different versions. Knowing the chronological order that the documents entered the Haystack might facilitate such a search.

As idea of “time versioning” [4], as I will refer to it in this thesis, was explored, several issues arose. The main issue was how to deal with the effect of deletion. We had to ponder the effect it would have on versioning if one or more versions of a document were removed from the Haystack. Moreover, if we were to allow a user to view an old version of a document, why not allow them to view an old version of their entire Haystack? Doing so would allow them a much more powerful way of searching for documents. If they could not locate a document in their Haystack, but could remember a time when they were able to, it would be nice to be able to repeat that search that was successful in the past. That idea became the basis for much of this thesis, and the idea of just content versioning itself was somewhat lost. The goal shifted to tracking change in one’s Haystack, and using that to facilitate using the Haystack. We started thinking of versioning from a broader perspective. Instead of
just trying to identify if a document entering a Haystack was already there in some form, we wanted to attack the broader problem of versioning in general. Tracking time in a logical manner became an objective. Also, the idea of change occurring in other ways was explored. While change can be seen as over time, was there other ways of viewing it? The answer was yes. In fact, it went back to the original idea. Change can be seen as a progression of states. (see Figure 1-2) In Figure 1-2, it can be seen how the addition of data moves the information repository from one state to the next. The repository is initially in state 1. When data X is added to the repository, it moves to state 2. Similarly, when data Y is added, it moves to state 3. State 3 is equal to state one, plus the addition of data X and data Y, in that order. Thus, while time is one way to measure the progression of state, it is not the only way. The dependence of those states on each other was another.

The notion of tracking the change in a Haystack in terms of the progression of states of the Haystack was called “data dependency versioning” [4], and will be referred to as such in this thesis. It is a slightly different idea than time versioning. Time versioning is the idea of observing when changes occur, and thus being able to simulate the state of the repository at any given time by knowing what was present at that time. Data dependency versioning is the idea of keeping track of what causes information to enter the Haystack. Thus, while using data dependency information could not tell you what your Haystack looked like last Tuesday, it could tell you why your resume is in it (you put it there!). By knowing why information entered the Haystack, whether it be the user or software acting autonomously, the user is given more power. If a search for documents in their repository about math is bringing up too many results, and they know they are looking for a paper that they wrote, they it would be useful to add that stipulation. More importantly, consider the case where a lot of bogus data suddenly appears in the repository. This could happen if software operating on the Haystack has a bug in it. It would be useful to know all the data that was added by that software, and to be able to search for it and remove the appropriate bogus data. Data dependency versioning would add that capability.
Figure 1-1: A time versioning tree for a document
Figure 1-2: A view of data dependency versioning
Chapter 2

Background

2.1 Necessary Knowledge

It would be useful for a reader of this thesis to be familiar with Java [5, 12, 11] as both the Haystack system and my thesis were implemented in java. If a reader is not familiar with Java, knowledge of object oriented design should suffice however. A reader of this thesis should also be familiar with basic graph theory [9]. Several references will be made to concepts from that area.

2.2 Other Research

A great amount of the work in information retrieval today is focused in the area of text retrieval. The main object in this research is fairly simple, though difficult to achieve. Given a corpus of documents, the goal is to be able to return the most relevant documents to a keyword search. This is the purpose of popular web search engines such as Google or Altavista.

There have been efforts to accomplish some of what is done in this thesis. People have made efforts to give the user the ability to view their information space the way it looked at any given point in time. Some people have tried to do this with a unique graphical user interface that allows the user to travel forwards and backwards through time visually in their information space. One such approach is described
in the December issue of “Technology Review” [14]. In this thesis, however, we do not tackle any of the GUI related issues with such a process. We merely provide an information repository that has the capability of being a back end for such an interface. People have also made efforts to track change in terms of data dependence. CVS [3] is a good example of this. CVS is most popular as a tool for source code control. CVS tracks who adds to or changes the information in the CVS repository, and allows a user to revert to a previous state if desired. The primary use of CVS is to allow a user to work on their own version of the repository, and then merge their changes, when desired, with the current version of the repository. The current version might be different from the one they were working with, particularly if other users have added changes of their own since they checked it out. I am not aware of any research that has been done that tries to combine time and data dependence to be even more useful to the user. The hope is that the combination of time and data dependency will provide an intuitive mechanism for an individual, based on the fact that most people can recall about memories either when they happened, or why they happened, but not always both.

### 2.3 RDF

RDF [6, 10] provides a basic method for storing information that can be understood by a variety of applications. The basic structure of an RDF statement is a triple. A

![Figure 2-1: An RDF triple - *statement*](image)

triple consists of three pieces of information which generally consist of two resources
(a resource is almost anything - it will be explained more a little further down) and a relationship between them. (see figure 2-1) Figure 2-1 Illustrates the statement “Apples have seeds” represented as a triple in RDF format. Each oval represents a resource, and the arrow represents the relationship between them. Though not immediately intuitive, it is possible to express most information in the form of one or a group of RDF triples. An example is given after describing the mechanics of RDF. Technically, a RDF triple consists of a resource, a property for that resource, and a value for the property. A resource can be almost any piece of information, in any number of formats, but is usually specified using a Universal Resource Indica
tor (URI). A URI is commonly expressed as a string of characters, which specify a protocol, a namespace, and an identifier within that namespace. A protocol is an indication of the method to use when communicating with the namespace. On the Web, for instance, this protocol is typically http. A namespace is used to state where the desired information is in general. An analogy would be a street address for an individual. If you are told to locate David Karger, but not told that he works at the Laboratory for Computer Science at MIT, the task would be very difficult. There could be many David Kagers in the world, and it would be impossible to ascertain which is the correct one. But by telling you that the namespace for David Karger is LCS at MIT, one can figure out that the person to contact is Professor David Karger at MIT.

The second element in the triple specifies the relationship, or property, that is relevant to the resource. The third element is the value for that property. In order to stay consistent with the terminology introduced in Ken McCracken’s thesis, as much of this thesis is built on top of the work done there, the elements of a triple will hereafter be referred to as the subject, predicate and object, respectively. RDF statements, as stated previously, can be combined to relate greater meaning. An example would be the best way to illustrate this. To start, consider a single statement that “apples have seeds”. This would be represented in RDF form with the subject being “apples”, the predicate being “have”, and the object being “seeds”. Now consider that it is desirable to state that “Johnny Appleseed says that apples
have seeds”. An additional RDF statement can be created with the subject being the entire prior statement. This is part of a process called reification that is just another way of saying “apples have seeds”. The triple (“apples”, ”have”, ”seeds”) becomes part of several other triples. For simplicity, let us represent that triple as *statement*. Reification involves creating several new triples that describe the aspects of “statement”. These triples would be (*statement*, “type”, “Statement”), (*statement*, “subject”, “apples”), (*statement*, “predicate”, “have”), and (*statement*, “object”, “seeds”). Adding the data that Johnny Appleseed says this is so then becomes the task of adding one more triple - (*statement*, “assertedBy”, “Johnny Appleseed”). These statements combined form the statement that “apples have seeds is asserted by Johnny Appleseed”, or in other words, “Johnny Appleseed says that apples have seeds”.

In this way, a user’s information can be broken down and stored as RDF statements, or triples. This is a useful way of storing a user’s information because it is a simple and widely used format. The hope is that the information stored in such a way can be accessible to other applications. Specifically, it would provide useful to
Haystack if Haystacks wish to communicate with each other. If one Haystack wishes to provide information to another Haystack, it is a easier process if that information is stored in the same format. Of course, Haystack could use any format it wished, as long as it kept that format uniform among Haystacks. The real advantage of using RDF is communication with other non-Haystack applications. If a Haystack wishes to retrieve information from the Web, RDF is a very convenient format for doing so because of a growing standardization around RDF [6, 10]. Moreover, if another existing application, or an application that is being designed, wants to communicate with Haystack or directly make use of its stored information, RDF provides an easy to use format that many software designers would already possess knowledge of.

Furthermore, RDF is useful because it allows the addition of namespace when specifying resources. While it might be fine for resources to be of local namespace when the user is adding information to the Haystack, it may not suffice when storing information from the Web, another user's Haystack, or providing information to an outside application. RDF allows the specification of a namespaces through the use of URI's. This is easily understood by anyone who has ever browsed the Web. In a Web browser one would usually type something of the form http://web.mit.edu/domdaleo/Public/Interface. In this form http specifies the protocol and web.mit.edu specifies the namespace in which to resolve domdaleo/Public/Interface.html. The power of this statement is that any person or application on any machine with Internet access can locate this piece of information. The statement contains all data that is necessary to do so. This is the power of URI's in RDF statements, and is one of the reasons RDF is chosen as the format for data entering the triple store. Thus, the important aspects of RDF is that it is easy to use, and perhaps more importantly widely used and accepted as a good basic method for storing information.

2.4 Haystack Data Model

With a basic understanding of how RDF works and is used, let us now discuss the Haystack data model. More importantly, we will see how RDF provides an easy way
to represent this data model.

The Haystack Data Model (HDM) [1] is elegant in design. It allows for a user's information to be stored uniformly in a simple, easy to understand manner. The main unit in the HDM is the straw. The Haystack Data Model is basically a graph, with each node holding some piece, or pieces, of information, and the edges between the nodes representing a specified relationship. A straw is a node in this graph. It can hold any amount of information, in any number of formats. To make the design easier to understand, several specific types of straws with set purposes exist.

![Diagram of the Haystack data model]

Figure 2-3: The Haystack data model

A Tie is an edge in the graph equivalent of the HDM. A tie has a specific meaning, as declared within each tie. In the same way that a predicate is the relationship between two resources, a tie represents the relationship between two straws. There are many types of ties. For a document, there will often be an author tie connected
to it. There could also be ties that point to the date the document was archived, or the key terms, and their frequencies, that should be associated with the document when performing a query. If a user manually adds metadata about a document, as they are allowed to do, a tie is created that points to that metadata.

A Needle is a straw meant to be a container for information. It might contain a text file that the user archived, the contents of a Web page, a string, or even just bits. There is no restriction on what may be contained in a Needle. The ties associated with a needle, and the information contained in the straws those ties point to will often give the information about what is contained in a needle and how to view it.

A Bale is a straw meant to be the center for a collection of related information. A document bale is a good example of what a bale is used for. Hanging off of a document bale will be needles that contain the text for the document, as well as pertinent metadata such as the author, or format of the document. In general, when something is archived into a Haystack, a bale will be created for it and all the information pertaining to that archive will be connected to the bale with the appropriate ties.

2.5 Haystack Service Model

With this graph model for the HDM, we now discuss the service model [2]. Services are used to perform actions on the graph, such as querying or archival. It is helpful to note that the naming convention for services is that they start with “Hs”. Thus, for example, the service that activates when a document is archived in the Haystack is called “HsArchive”. Services can be triggered by the user, but can also be set up to trigger on their own. When the user archives some piece of data, the archival service, HsArchive triggers. That service is responsible for setting up a Bale for the piece of information, and often creating a tie and needle for the body of the text. Other services will trigger off their specific star-graph. A star graph is a structure in the HDM graph that a service will operate on.

Staying with the example of a user archiving a document, after HsArchive creates
a document bale, and a tie and needle for the body, this will trigger HsTypeGuesser, the type guessing service. Let's assume that the document the user was archiving was a BABYL file. A BABYL file is a text file which is a concatenation of text emails, with markers between the emails so the individual emails can be identified. The star graph for HsTypeGuesser is merely a document bale with a "body" tie pointing at a needle that contains the body. HsTypeGuesser will generally trigger on all documents that are archived, once that structure is established. HsTypeGuesser then examines the body of the document, and tries to guess the type, or format, of the document. In this example, HsTypeGuesser will establish that this is a BABYL file and create a "type" tie from the document bale to a needle that contains the information that the type is BABYL. This will in turn trigger HsBABYL. The star graph for HsBABYL is a document bale with a "body" tie pointing to the body and a "type" tie pointing to a needle that specifies the type as BABYL. HsBABYL will read the body of the BABYL file and isolate each individual email. For each email, it will create a "contains" tie between the document bale and a new document bale, which it creates, for that individual email. Thus it can be seen how the Haystack service model is set up, and how it leads to the creation of the graph that is the HDM.

RDF is a good way of representing the HDM for several reasons. The HDM revolves around pieces of information and the relationship between them. RDF is set up in a similar manner, with the use of triples. Moreover, as the HDM is a flexible graph that is often changing, RDF can easily used to represent these constant changes with the addition of more triples.

2.6 The Ordinary Triple Store (a.k.a. Ken Mc-Cracken's triple store) and the Interface to it

We now look at the basic features of Ken's triple store, hereafter referred to as the "ordinary" or "plain" triple store, in contrast to mine which will be referred to as the "fancy" triple store, that are vital to my system. The work that is done in this thesis
relies on the existence of an underlying triple store to contain both the information that a user wishes to put into the “fancy” triple store, as well as the data needed to track time and data dependency. Fortunately, as this thesis was designed, Ken McCracken, also a MEng student working on the Haystack Project, designed and implemented a basic triple store that would meet the needs of this thesis. For a more detailed description of Ken’s triple store, refer to Ken’s thesis, “Flexible and Robust Data Storage and Retrieval in the Haystack System” [8].

```java
public void beginTransaction() throws NoTransactionException
public void commitTransaction() throws NoTransactionException
public void abortTransaction() throws NoTransactionException
public URI putAssertion(URI subject, URI predicate, URI object)
public void putBits(URI uri, Serializable theBits)
public URI getSubject(URI rdfStatement)
public URI getPredicate(URI rdfStatement)
public URI getObject(URI rdfStatement)
public Serializable getBits(URI bitsKey)
public Enumeration getAssertionsBySubject(URI subject)
public Enumeration getAssertionsByPredicate(URI predicate)
public Enumeration getAssertionsForObject(URI object)
public Enumeration getAllAssertions()
public void expunge(URI uri) throws AbortException, NoTransactionException
```

Figure 2.3.1 - A brief look at the ordinary triple store interface
The ordinary triple store provides several basic functions that are needed to construct the system described in this thesis. (see Figure 2.3.1) The ability to perform puts and gets is obviously vital. A put is the procedure that adds information to the repository, and a get is the function that retrieves information from the repository. It is convenient that the plain triple store is also based on storing information in an RDF format. The interface to the ordinary triple store is consistent with this approach and proved easy to use. While it might be possible to implement my system on top of a data store that did not make use of RDF, it would likely be less efficient and increasingly difficult to understand. The ability to perform an expunge on data is in addition quite important. Expunging data is the process of irrevocably removing that data from the repository. A very useful aspect of tracking time and data dependency information is that it provides greater flexibility and power to undo possibly traumatic events in your information repository. An example of this would be if a program, or service, that is operating autonomously on your information runs rampant and adds bogus data. Without the ability to expunge bogus data, power would be lost.

Another key aspect of the ordinary triple store is that it provided a transaction management system consistent with the ACID philosophy. It is not important to understand exactly what the ACID philosophy is right now, simply that the plain triple store adheres to the ACID guidelines. Transactions are implemented as described below in the section labeled “Transaction Management”. This allowed me to build a system on top of the plain triple store that also provided ACID transactions. If another triple store was to be used as the base for the system I have created here, it is essential that it also adhere to the ACID guidelines.

2.7 Transaction Management

A transaction is a collection of operations on, in this case, the plain triple store (see section labeled 'The Ordinary Triple Store'). There are several objectives when designing transactions, with the goal being that the data in the triple store is main-
tained in a consistent state, even if the hardware or software crashes. The user has
the ability to start a transaction, perform operations, and then commit or abort a
transaction with the assurance that whatever is contained within the transaction will
either all happen, or not at all. It goes without saying that maintaining uncorrupted
data that can always be recovered regardless of hardware failures is very desirable.

To understand this fully, it is vital to discuss those properties which are essential
to transaction management, namely the ACID properties. If one adheres to the
ACID properties when designing transactions, the issue of maintaining consistent
and correct information is reduced to doing so. The ACID properties are Atomicity,
Consistency, Isolation and Durability. Atomicity means that the operations that
make up a transactions either all occur, or none at all. This is important when
considering recoverability. If the system crashes at any point, only those transactions
that have been completed and committed will result in permanent changes when
recovering. Thus the user can know what to expect when they restart their system,
and they know when changes become permanent. This gives the added ability to
abort transactions and avoid permanent changes. Consistency means that if the
state before a transaction was consistent with certain constraints, the state transition
made by the transaction leaves the repository consistent with the same constraints.
This insures that the information will always be consistent, if it begins in a consistent
state. Isolation is important when considering parallel operations. It is possible that
transactions may be executed in parallel, and when this happens, one would like
the composite effects to be the same regardless of the order in which they occur.
If one adds two separate pieces of information to their repository, the net effect on
the repository should be the same regardless of which order they are added in - the
repository should now contain both those pieces of information. Durability means that
any transaction which is completed will survive a system failure. This is important
for obvious reasons.

Committing a transaction is the exactly what the name implies. When a transac-
tion is committed, the effects of the operations contained within it become permanent.
If a transaction is instead aborted, the effects of that transaction are reversed, as if
the transaction never occurred. Thus, recovery simply becomes the process of undoing any transactions that were in process, but were not committed before the system crashed. This ensures that the system is still in a consistent and correct state after recovery.
Figure 2-4: The Star Graph for HsTypeGuesser
Figure 2-5: A BABYL document in the HDM
Chapter 3

Design

3.1 Design Overview

While the design of this thesis went through many stages, it eventually settled on the idea of having two separate versioning layers. Combined, these two layers would form the fancy triple store. The time versioning layer would be on the bottom, and use a plain triple store where all information would ultimately be stored. That information includes both the data the user wishes to store, as well as the data generated by these two layers for their own use. The data dependency layer would be on top of the time layer, and make calls to the methods of the time layer. Thus, a user of the fancy triple store is actually making calls to the data dependency layer.

To fully understand the functions performed by each layer, and how these combine to be the fancy triple store, let us now examine the requirements of each layer.

3.1.1 Time Versioning Requirements

1. Be able to roll back the clock and..

    (a) do queries on the past, over an interval of time or at a specific point.

    (b) view and browse the Haystack the way it looked at that time.

2. Be able to revert to an old version (rewind)
3.1.2 Data Dependency Versioning Requirements

1. To be able to undo an archive and its consequences

2. To be able to undo a service’s activities
   (a) All of a service’s activities
   (b) All activities from a certain point (time) on
   (c) All activities after a certain archive

3.1.3 Evolution of Design Requirements

Initially, there was one requirement to fulfill. We wanted to design a system that, given an input to the information repository, would be able to identify whether that information was already present in the repository. That information could either be in the form provided, or possibly in some other form such as an older version.

The questions that we set out to answer were the following:

1. When is something archived decided to be a new version of something already in the repository?
   (a) Is it easier to decide this depending on the content (i.e. text, bytes, web page)?
   (b) What if the change is small? What constitutes a big enough change to be considered a new version? (An example of a change that perhaps should not be considered a new version is fixing a single character typo)

2. If something is a new version, how does the meta-data from the old version apply to the new one?

3. Similarly, should meta data added to a new version apply to an old version?

4. What if meta data suggests that a supposedly new version is in fact not a new version?
5. Should old versions be seen by the user, or do they lose usefulness once there is a new version?

6. Should querying old versions be allowed and if so, how?

As the issues surrounding these questions were explored, several other issues came into the light. If older versions of information being passed into the repository were to be identified, it would be useful to keep older version and connect the newest version to the older versions so that the user can find them more easily. It is very likely that a user will not like losing an old version of something just because they archive a new one. The case could certainly arise where a user might not be satisfied with the newest version of something and want to revert to an older version. But if the user is given that ability, then why not give them the ability to view any old information? In fact, why not give the user the ability to view the state of their information repository at any point in time? It was decided that this should be a requirement of a versioning layer, with the details of exactly how to be fleshed out.

As further discussion of the topic was held, the question was posed as to what a versioning layer would be useful for, and therefore what it should be capable of. One such use would be to identify the relationships between old and new versions of the same information. An example of this is a composition in the many stages of revision. Another would be to allow the user to not only view the state of their information at any point in time, but also to query their information in that state.

As stated earlier, the requirements of this versioning layer, which became known as the time versioning layer, were outlined as follows:

1. Be able to roll back the clock and...

   (a) do queries on the past, over an interval of time or at a specific point.

   (b) view the Haystack the way it looked at that time.

2. Be able to revert to an old version (rewind)

As time progressed, the idea of a versioning layer began to evolve. If the user was to have the ability to view their information at any point in time, and to know
when data entered or left their repository, then it would be useful to know why that information was there. One such use of this would be if a malicious or buggy entity added a lot of incorrect information to the repository. It would be nice to be able to remove all such data. Thus the idea was born of data dependency versioning. It is easy to envision the way data creates versions along a time line. Version A of the repository spawns version B as time progresses and items x, y and z are added to the repository. But think about changes in terms of why they happen. That is how versioning can be seen from a data dependency perspective. Version A has item x, and item x leads to the creation of item y, which is now part of a new version, version B. Thus, versioning is just a way of ordering the states that the repository achieves. Ordering those states by time is one such way. However, ordering those states by their dependence on each other is another such way. Giving the user information about why data is in their repository therefore becomes an objective.

As stated earlier, the requirement of such a versioning layer, which became known as the data dependency layer, were outlined as follows:

1. To be able to undo an archive and its consequences

2. To be able to undo a service’s activities

   (a) All of a service’s activities

   (b) All activities from a certain point (time) on

   (c) All activities after a certain archive

With the objective to track changes in terms of time and dependency, this thesis took on a new meaning. The initial goal of being able to identify whether a given piece of information existed in the repository in some shape or form took a back seat to the ideas of time and dependency. In the end, that goal would be left as a possibility for future work, as the ability to track change took center stage. It would seem that given that ability, the initial goal might be somewhat easier, but it was still beyond the extent of this thesis to achieve it, mostly due to time constraints.
3.2 Time

Time is an important way of tracking change. Many aspects of life revolve around time. People naturally use time as a way to track the change in their lives, and in themselves. Birthdays, anniversaries, and leases are just a few examples. All these things mark significant change in our lives, and use time to benchmark it. Our memories are not just a random jumble of information in our heads. Rather, they are often easily organized into the order that they happened. And even if they cannot be ordered exactly, they can often be ordered more generally, with an individual being able to recall that an event in their life occurred during a specific time period, if the specific instant cannot be remembered. The Lifestreams project operates on this principle. Lifestreams is an information management model for an operating system that organizes all of a users information into a timeline. When a new document enters the system, it is attached to the end, or the portion in the present, of the timeline. All documents are fully indexed and can be searched by content or descriptive metadata. The hope with Lifestreams, as with the work in this thesis, is that allowing a user to search by content, and also along a timeline, will provide an intuitive interface to their information.

Thus an argument can be made for using time as a way of tracking change in our information. Since people feel comfortable tracking changes in their lives in terms of time, the hope is that they will also feel comfortable with the notion of tracking their information, and its changes, in the same way. If information that an individual once possessed in their repository cannot be found in the present, perhaps they can remember when it did exist and search for it then. Or in the worst case, they can recall a time period that it definitely existed, and look for it over that interval. Much the same way that we can remember events in our own life, we can remember events in the life of our information, and use those memories to keep it better organized. With the events in our life, we can merely reflect on them. We can use those memories in the present if they prove useful. Remembering someone’s birthday, so you can mail them a card, by recalling their birthday party (and the day it was on) from a few years ago
that sticks in your head more than the actual date, is one such example. We cannot, however, go back and change those memories. That would require a time machine, something the human race has yet to accomplish. And even if a time machine did exist, who knows what effect going back in time and fiddling with events would have. The idea of time travel raises many issues that cannot be easily answered.

3.2.1 Time Machine

The first idea for allowing the user to view their information the way it existed in some past state was to give them an information time machine. They could travel back into information time and see what things looked like. They could search for information there, and if found, alter it. Alternatively, it could be put into the time machine, and taken back to the present for manipulating. If this model were to be used, several questions would have to be answered. What would the effect of altering information in the past be on the present state of the repository? That might be figurable if one only ever traveled back to one point in time to make changes. But what if multiple trips were made? If a user traveled back to time Y, made some changes, and came back to the present, the present would certainly be different. Imagine then that the user went back to time X, X before Y, and made changes that would affect the changes that were made at Y. Now the question arises as to what actually happened at Y. It certainly would not be what was remembered by the user, because the state of the information that user dealt with when they went back to state Y would not have been the same if they had gone back to X first. Furthermore, the present would also be different, though it would be near impossible to say how. For these reasons, the model of a time machine for viewing past states of the information repository was deemed poor.

3.2.2 Crystal Ball

Making a model that is more consistent with the way an individual can access and use their memories is a better idea. Imagine an information crystal ball that allows
the user to view their repository the way it looked at any point in time, and search for information at any point, or over an interval, in the past. Information that is found in the past can be virtually pulled into the present for manipulating by making a copy of the information in the present. This gives the same power that a time machine model would, without the complications. A user can access their information in the past, and use their memories of experiences with that past information to more efficiently search. Moreover, if they need information in the present that only existed in the past, they can “bring” it there, and use it.

3.2.3 Rewind

The user is given, for extenuating circumstances, a limited time machine. A rewind mechanism would be a better analogy, actually. A mechanism that only allows the user to travel into the past, and then traps them there. This is useful in case terrible things have happened, and the user wishes to rewind back to a time before those events and prevent them. In the movie “Terminator 2”, this is the idea that is used. The present depicts a holocaustic world where machines rule and humans form a resistance to fight for their lives on a daily basis. The machines send a terminator back in time to try and kill the human that started the resistance. A similar circumstance could be imagined with one’s information. If a service runs rampant, and destroys much information (creating a holocaustic information world), it would be useful to travel back before that service existed, or perhaps before it ran rampant, and prevent the same thing from happening.

3.3 Dependency

Why things change is also an important way of tracking change. Often, if one cannot remember when something occurred in their life, they can certainly remember why it happened. As one gets older, this becomes even more so. With more memories cluttering each other, it may become difficult to recall the order. Therefore, it is also vital to keep track of why things change. It provides a very useful complement to
the idea of when things change when it comes to searching for information. A real life example might be an old girlfriend. Perhaps one cannot remember when they dated a particular girl, but they can remember that they were introduced by their roommate. The same can be said for an information repository. Perhaps one desires information that existed in the past, but they cannot remember when. If one can remember why that information entered the repository, it will be possible to find it. Moreover, it might also be remembered who (I use who here to mean more than a person - possibly also a machine, or software) put that information in the repository. An example would be if a user was trying to find a recipe. The may not be able to remember when they put that recipe into their repository. If they can remember that their mother gave them the recipe, then they know why it entered the repository and they can find it.

3.3.1 WhoCreated

The first thing to keep track of is who adds something to the repository. There could be several sources for this. The user has the ability add things directly, so they are a possible cause. Software, in the form of services, could also be the reason for information being added. That information could be extracted from data that already existed, or it could be brought in from the outside, such as the Web. Haystack itself could add information to the repository, as is the case when the time and data dependency layers add information that is used to track change. In most cases, there will only be one WhoCreated for a given piece of information. An exception to this would be if data is added to the repository that is already there. In this case, the entity trying to add the data again will be recorded as another WhoCreated. The reason this is necessary is discussed later, in the section on the propagation of deletion, and undeletion.
3.3.2 WhyCreated

It is also vital to keep track of why data enters the triple store. This is a slightly different notion than who is responsible, though in some cases the answer might be the same. If a service that extracts metadata from emails in the repository adds a needle with the author, the who and why are very different. The WhyCreated is obviously the service doing the adding. The WhyCreated, though, is the email itself. Without the email, the service would have never been triggered because the archiving of the email creates the necessary star graph. The email is why the author needle was created. Without the email, it may in fact be the case that the author needle has no meaning. This comes into play later when discussing deletion, and the need for propagating deletion. Unlike WhoCreated, it is not fair to say here that there will often be just one WhyCreated for each piece of information. More often than not, a service triggers on the existence of multiple pieces of data, so each of those must be considered a WhyCreated.

3.3.3 Time Reliance

It is important to note here that the data dependency layer is somewhat reliant on the time layer, while the opposite is not so. This is mostly due to issues that arise with deletion. I will not discuss it in full here (I discuss it in the next section), but I bring up the main issue. Consider data X in the repository. Data Y and data Z list X, and only X, as WhyCreated. Thus, if X were to be removed, Y and Z should also be removed. I discuss and justify this in the next section. But what if I merely want to say, delete all that X is responsible for the existence of (all those items that have X, and only X, as WhyCreated). If I do this in the present, it seems simple enough, delete Y and Z. It would be very useful, however, to use a point in the past as a starting point. In other words, to delete everything that X is responsible for, but only after time t. If Y existed before time t, and Z after, in this case only Z is deleted. This operation is not possible, however, if the notion of time does not exist. Thus, the dependency layer is somewhat reliant on the time layer to provide
full functionality. An example where this might be desired is if a service that has been working fine suddenly breaks. The service can be fixed, but what about all the incorrect information it added to the store while it was broken? It would be nice to say, delete everything that has that service as a WhoCreated, but only after time t, the time it broke. This is clearly a desirable function of the data dependency layer, but requires the existence of a time layer.

3.4 Deletion

The idea of deletion is a basic one in regards to an information repository. If the user has the ability to add information, they must certainly have the ability to remove it. The question becomes, then, should the user have the ability to permanently remove information from the repository?

The amount of discussion on this topic was surprising. It was not immediately clear what way of doing this would be intuitive to the user. It was clear, however, that a design that a user would find intuitive was very desirable. The removal of information from the repository is a serious issue, especially if the user was to be given the ability to permanently remove information.

The first idea that we came up with was to give the user the option of deleting anything in their repository, but not permanently. Much as in Windows when a user deletes something and it goes to the Recycle Bin where it can later be retrieved from if the user changes their mind, we wanted to give the user the ability to later undelete something they had deleted. In essence, when they deleted something, it would remain in the repository but become hidden so that from their perspective it would seem like it was gone. If they decided later that they wanted to restore it, they could.

Giving the user this function seemed like a good idea. The next notion that arose was to give the user a way to explicitly hide things. The idea behind it was that the term “hide” was clearer as to what was going on. Hide would do the same thing as delete, but would carry a different connotation. From a user’s perspective, if they
felt like they never wanted to see a particular piece of information again, they could delete it. If the felt like they just didn’t want to see it right now, they could hide it. Both operations would be undoable.

This still left out the ability to truly remove data from the repository though. There was some debate as to whether delete should be permanent. This was decided against because it would be counterintuitive to what any user of Windows was used to. It is hard to ignore the way that Windows has shaped people’s views on computing.

After discussing hide and delete for a while, the idea was proposed to include a third option, expunge. Expunge would remove information from the repository permanently. It would not be reversible. It was felt that this was necessary for several things. If there was a bug in a program running on the repository, it would be nice to be able to remove information that was bogus. Moreover, unless space was unlimited, a user might need to free up space.

Finally, the decision was made not to have hide. Since it would only really be doing the same thing as delete, it didn’t provide extra functionality. If anything, it seemed to add an extra degree of confusion. It is easy to envision a user trying to figure out what the difference between the two operations is because it isn’t logical that they would do the exact same thing. It would be good to avoid a situation where a user might be unsure as to which operation they want to perform.

Going back to the question of whether a user should have the ability to permanently remove information from the repository, it was thus decided that it was important that the answer to this question be yes, but only to a certain extent. Everyone, at one time or another, has deleted something that they later wish they had not. This is the reason that Windows, for example, has a recycle bin where items that have been ‘deleted’ can be retrieved later. In effect, those items were only really hidden from the user. To fully delete them, the user must go to the recycle bin and delete them again, or just empty everything in the recycle bin. It was felt that this was a good policy, and a distinction should be made between items that were being deleted, and could be later ‘undeleted’, and those items that would truly be gone forever. The hope is that in most cases the user will merely choose the less serious
option, saving true expunge for special circumstances.

Therefore, the user has two options when it is desirable to remove a piece of information from the repository. The first is 'delete'. The actual effect of a delete is that triple store will keep track of when that item was deleted, and make it invisible to the user. If the user ever wants to restore that item, they can use the 'undelete' function, which will also be available. An example of when this might occur is if the user is querying on the past, and they find an item there that is deleted in the present, but they would like to have it back. In this situation, they can simply undelete it in the present, and then manipulate as they choose. The other option that the user will have is 'expunge'. Expunge is much more serious, and it should be made clear to the user that this is so. When an item is expunged from the data store, it will be irrevocably gone. It will not be possible to query on the past, using the 'crystal ball', and find this item. It is expunged from existence. Though this is a vital function, we expect it will be used infrequently. It’s main usage would be to remove incorrect or embarrassing information that might be added to the repository by broken services.

It is possible, therefore, that a piece of data might be visible over intervals of time. If the user adds data X at time t1, deletes it at time t2, undeletes it at time t3, then deletes it again at time t4, that item will be visible to the user whenever they are viewing the state of the repository during any time between t1 and t2, and between t3 and t4. This could obviously be extended to any number of intervals. An alternative to this approach, which allows time intervals, was to state the status of an object, in the present, as simply deleted or not. Thus, an object could have a deletion time, and it would be deleted from that time up until the present. If it was undeleted, it would exist again in the present and the undeletion time would be forgotten. The advantage to this approach is its simplicity. There would be less information storage needed to keep track of deletion. Moreover, the argument could be made that if the user undeletes something, it might be possible to assume that the data is important to the user, and therefore it is useless to have that item deleted during any period of time. However, there were also problems with that approach, which outweighed the advantages. It would be nice to the user, when browsing through past versions of
their data, to have things appear the way they looked at that time. With this deletion approach, they would not necessarily. Items might appear in past versions, though they were deleted at the time. It seems nicer to keep it intuitive, so that the user's memories would actually aid them in the search process. Furthermore, if an item is undeleted, then deleted again, the argument about assuming that an undeleted item is important to the user and therefore should appear in the past though it was deleted is no longer valid. And when that item is undeleted again, it would be impossible to regain that information about the first interval that it was deleted. For these reasons, it was decided that items could be deleted over multiple intervals of time, and present between those intervals. This is important to understand for the idea of searching an interval of the past. When the user issues a query over the time period t1-t2, the results will be filtered to include only that data that existed (was in the triple store and not deleted) during some point in the interval. It is not necessary that the data exist over the whole interval, merely at some point. This is chosen because it will make search more useful to the user by increasing the usefulness of searching over an interval. If one can merely remember some time period when one had access to some data, which one apparently does not anymore in the present, one can find it. If one chooses the other alternative, one would have to remember a time interval during which the item existed at every point, a much more difficult thing to do. For an

![Diagram](image-url)

**Figure 3-1: A search interval and the periods that a piece of data exists**

example, see figure 3-1. In this figure, one can see the life of a piece of data. During
the middle of the life of the piece of data, it was deleted, and then later undeleted. The query indicated, in this case, would return that piece of data because it existed for a portion of the query interval. It is important to notice that the data only existed for part of the search interval, not the entire interval, yet it is still returned.

The final aspect of deletion is the propagation of deletion. Note that everything said here is also true for expunging, as propagation of expunging will be handled the same way. The simplest way to handle deletion would be to just delete those items that the user requests. However, it would be much more useful if, when item X is deleted from the repository, that all other data that simply has no meaning or purpose is also deleted. This would prevent a user’s information space from getting cluttered with information that is useless to that user. Information that is only useful when combined with other information in the repository, and is only in the repository because of that other information, falls into this category. The way it is decided what information has no meaning or purpose without item X is a simple process. Any data that has X as its WhyCreated, and no other item as its WhyCreated, would also be deleted. Furthermore, if those items are called S, any data that only has items from S as its WhyCreated would also be deleted, and so on, until the propagation is complete.

For a clearer picture, refer to figure 3-2. In this figure are depicted several pieces of data. An arrow represents a dependency. Thus, for example, Data D is dependent on Data A, and only Data A. Data C is dependent on Data A and Data B. If Data A is deleted in this case, it follows that Data D, Data E, and Data G will also be deleted by propogation. Since Data D only depends on A, it is deleted by propogation first. Data E, depending only on Data D, is then deleted also. Data G, depending only on data E, is then deleted. The other pieces of data do not get deleted because they depend on data that was not deleted. Data C, for example, is not deleted even though Data A is deleted, because it also depends on Data B, which still exists.
Figure 3-2: A view of several pieces of data and the dependencies between them
Chapter 4

Interface

4.1 Ken’s interface note

A reader of this section should be familiar with the basics of the interface to Ken McCracken’s triple store [8], the plain triple store, which is reviewed in section 2.6. The plain triple store follows a transaction model exactly as I have described it in the Background section. The plain triple store also relies on information in an rdf format. For more information on the interface to the plain triple store, refer to Ken McCracken’s thesis [8].

4.2 Getting Started with the fancy triple store

For the fancy triple store, certain aspects of the interface will remain the same as the plain triple store. The user will handle transactions in exactly the same way, using the same methods as with the ordinary triple store (see appendix A). All methods that were available in the ordinary triple store interface will also be available to the user of the fancy triple store, and will perform the same functions. This provides the convenience of being able to treat the fancy triple store as ordinary, if that is desirable.

Connecting to the fancy triple store will likewise be handled in the same way as with the plain triple store. The user has the option of either connecting directly,
or remotely. The constructor can be called with no arguments, which chooses the
default table in the database to store information, or with a string that specifies the
name of a table to use in the database. The purpose of this is to allow the user
to have multiple fancy triple stores operating at the same time, and store separate
information in each of them.

4.3 Using the fancy triple store

The methods that will be embellished include the put, get, and expunge methods.
Entirely new to the fancy triple store interface will be methods to perform deletion,
and when necessary, undeletion.

Puts must be in one of the following two parameter formats:

1. Just the information that is being put into the TS will be passed in. The
time layer will itself generate information about the CreationTime. The format
for information being put into the triple store can be in one of two forms: a
(subject, object, predicate) assertion, or a collection of bits.

2. The format from 1, plus parameters for WhyCreated and WhoCreated. Either
of these arguments may be passed in as NIL if it is desirable to not specify one
of them.

It is important to note that the only way to get the WhoCreated and WhyCreated
information into the repository is to use the second format, and specify it at the time
that data is inserted. Most of the functions provided by the data dependency layer
require that this is done. While this creates more overhead work for the user of the
fancy triple store, there are not many other options. If the user is not specifying that
information, there would have to be another source for it, an oracle of sorts. But
the user would unavoidably have to specify, and likely therefore create that oracle,
possibly generating even more overhead work.

Gets must be in one of the following six parameter formats:
1. Just the information necessary to specify what is desired out of the triple store. That can be in one of five formats:

   (a) An individual assertion. When this is the parameter, the user can request the subject URI, object URI, predicate URI, bits, or local URI associated with the given assertion.

   (b) Nothing. In this case, all assertions in the triple store are returned.

   (c) A subject. All assertions with the given subject are returned.

   (d) An object. All assertions with the given object are returned.

   (e) A predicate. All assertions with the given predicate are returned.

2. The format from 1, plus two times (t1,t2) specifying an interval. By passing in t1 and t2 equal to one another, the user can specify an instant. It is also valid to specify t1 as “START”, which refers to the time that the triple store came into existence. All data in the triple store will have a CreationTime greater than START. In addition, either time can be passed in as “PRESENT”, which means exactly what the name implies. Any information that existed at any point during the time interval, and was not deleted at that point, is eligible to be returned, given that it also matches the information specified in the format from 1.

3. The format from 2, plus a parameter for WhyCreated. This filters the results the format from 2 would return to include only those pieces of data that have the correct WhyCreated clause associated with them.

4. The format from 2, plus a parameter for WhoCreated. This works in the same way.

5. The format from 1, plus a parameter for WhyCreated. This works in the same way.

6. The format from 1, plus a parameter for WhoCreated. This works in the same way.
Contains methods will stay exactly the same. There is no useful way to add time or data dependency information to these methods, so they will be available to the user in the same way as in the ordinary Triple Store.

Expunge can be used in only one format.

1. Just the information dictating what is to be expunged from the Triple Store. That will be either a (subject, object, predicate) assertion, or a collection of bits.

It seems obvious that situations will arise where the user will wish to rewind to a former state of the information repository, or undo bogus data added by a broken service. Expunge will likely be used for this. To perform these actions, it should be used in conjunction with the get methods. The get methods would be used to retrieve the information that needs to be expunged, and then expunge would be called with the results.

Deletes and Undeletes can be used in the same way, in only one format.

1. Just the information that is to be deleted from the Triple Store. That will be either a (subject, object, predicate) assertion, or a collection of bits.

It may also seem obvious that a situation will arise where a user wishes to delete all data with a particular WhyCreated, WhoCreated, CreationTime, or range of CreationTimes. To perform these actions, delete should be used in conjunction with get methods. The get methods would be used to retrieve the information that needs to be deleted, and the delete would be called with the results.

4.4 The Evolution of the Interface

The first idea that we had for how to handle the issue of turning back the clock was to rely on the GUI. The GUI would be expected to allow the user to specify a time at which to view their triple store, with the present being the default. That time would
simply be stored in one place, and any methods that needed to know it would access that place.

Setting the time in just one place didn’t seem like a very good idea. It would limit the user in that they would have to wait for queries on the past to complete before they could perform other queries on different time periods. If they didn’t, they might get incorrect results from the previous query by changing the global query time. It was therefore decided that the time desired for queries and gets should be passed into the appropriate methods.

The next dilemma that I had to consider was how to structure the different layers of the fancy triple store. I had to answer the following questions:

1. Should time and data dependency be in one layer, or separate layers?

2. If they are separate layers, which layer should be on top?

The first question I answered by deciding on separate layers. It would be better to make the design modular so that modifications to one aspect of versioning would not necessarily affect the other. As far as the second question, I decided that the time layer should be below the data dependency layer. This was for one reason. One of the main things that the data dependency layer should be able to do is give the user the power to undo the actions of a broken service. Along those lines, a user would likely either want to undo all the actions of a broken service, or only those actions after a certain time. To perform the latter, the data dependency layer needs the help of the time layer. While the same functionality could be provided with the data dependency layer below the time layer, it would require the time layer to have methods that were primarily about data dependency. It was my feeling that methods to undo the actions of a service after a certain time belonged in the data dependency layer. Moreover, this configuration seemed good because the time layer did not need the data dependency layer to perform any of its primary functions. The time layer’s primary functions only involve the data that the user stores, as well as CreationTime, DeletionTime, and UnDeletionTime.
With the layers positioned this way, it was easy to have the time layer (class) extend the plain triple store (class), and have the data dependency layer (class) extend the time layer (class) so that calls could be made directly to any of the methods in any of the layers (classes). This is desirable if, say, the user just wants to query their Haystack. It wouldn't make sense to have the user call a method in the data dependency layer (class) which would then call the method they actually want. Using Java makes this a trivial matter.
Chapter 5

Implementation

In the implementation of this thesis, it was decided to make two layers. One layer would be for time and the other would be for data dependency. These layers would need a repository for information. The choice was Ken McCracken’s triple store /citekens-thesis. Two of these repositories were utilized in the final implementation. One stored all the information that existed in the present state of the fancy triple store, and the other stored all information that had been deleted at some point in the past. The time layer rests directly on top of these two plain triple stores, and the data dependency layer on top of that. In essence, the fancy triple store is the data dependency layer. We now take a look at how each of these layers was implemented, starting at the top and working down.

5.1 Data-dependency Layer

The data dependency layer is implemented making use of one time triple store (a.k.a. the time layer). Any calls to methods of the data dependency layer either make one, or several calls to the time layer. No method calls are made directly to the underlying basic triple store. Invocation of put methods will call the put method of the time layer with the information being stored. It will also call the put method of the time layer to store WhoCreated and/or WhyCreated when appropriate. Get methods will call the appropriate get method of the time layer, based on the inclusion/exclusion
of time information, and then filter those results if WhoCreated or WhyCreated was specified. Deletion/undeletion and expunge calls will call the method of the same name in the time layer. In addition, get calls will be made to the time layer to determine if the effect needs to be propagated, and how far. This is explained in more detail below.

5.2 Time Layer

The time layer is implemented making use of two basic triple stores. The function of the first, the “present” triple store, is to store all assertions concerning subjects that exist in the present information state. By exist, I mean all that information which is in the triple store, and enjoys undeleted status in the present state. All other assertions are placed in the “past” triple store. This store holds data concerning all subjects that are deleted in the present state. The reason for this is to make queries on the present state of the triple store most efficient. The user will likely be operating on that present state most of the time, so it makes sense to optimize the fancy triple store with that in mind. The ServerPreferences file is used to specify what table names should be used when instantiating these ordinary triple stores. The user has the ability to alter these, though the default is table names “Past” and “Present”. The ServerPreferences file also has information concerning what database should be used, and therefore what database those tables will be found in. This information is less important to the fancy triple store, as the basic triple store is accountable for retrieving stored information. If the user does not alter the specification in the ServerPreferences file, it is unlikely this will be a problem.

The user has two options when connecting to the fancy triple store. They can connect remotely, if desired. This requires that the fancy triple store is running on a machine somewhere, and can be done in the same method as connecting remotely to the ordinary triple store (see Background section). If this is not the case, the user can connect directly to the fancy triple store by instantiating it. When instantiating the fancy triple store, it must also be initialized. The initialization caused the data
dependency layer to initialize a time layer, which initializes two ordinary triple stores, according to the information in the ServerPreferences file.

5.3 Transactions

Beginning a transaction on the fancy triple store is very similar to the same action on the ordinary triple store. From the time that a transaction is opened, no data put in is permanent until the transaction is committed. This allows the user to abort and undo anything that was part of the current transaction. The way this is implemented involves starting multiple transactions in some cases, though. A transaction started on the fancy triple store will always involve starting just one transaction on the data dependency layer. It will likewise involve starting exactly one transaction on the time layer. The time layer, however, may need to start multiple transactions, if it needs to operate on both the past triple store and present triple store. The way this is kept atomic is fairly simple and will be explained.

If information is being moved from the present triple store to the past triple store, in the case of a delete, the transaction on the past triple is opened and performed first. This is so that if the transaction is aborted from the user level, it is recoverable. The information can simply be removed from the past triple store by aborting that transaction. After the information is added to the past triple store, but before it is committed, a transaction is began on the present triple store, and the data is removed from it. At any time up to this point, if the entire transaction is aborted from the user level, everything can be undone by simply aborting both transactions on the ordinary triple stores. A commit action by the user involves invoking a commit transaction on the data dependency layer, which in turn commits the transaction on the time layer, and finally commits the transaction on either or both of the ordinary triple stores, as necessary. In any case, the information will never be lost because it is added to one triple store before it is removed from the other. This is important in the case of a system crash. If such an event occurred, no information would be lost. The only adverse affect that could occur would be that the present triple store might contain
information that is actually deleted. The way this is handled is described below. Information is also never lost in the case of an undelete, where information will be added to the present triple store first, and then removed from the past triple store. It is important to note that information could end up in both triple stores. If the data is committed to the past triple store before it is removed from the present, or vice-versa, and then something happens such as an abort, this would occur. In the case that this happens, it is not a problem because the information still exists to figure out when the data existed. Querying the past triple store will thus cause not problems. The only problem might be if information that is actually deleted in the present is in the present triple store. However, this is solved by doing a simple check before returning information from the present triple store to insure that information is not deleted.

If the operation being performed is a put, a transaction that is started simply trickles down and results in a transaction being started on the present triple store. Likewise, commits and aborts from the user level will trickle down and eventually be performed on the present plain triple store.

5.4 Put operations

When the user performs a put on the fancy triple store, they have the option of specifying either WhoCreated or WhyCreated, or both. This is totally up to the user, and though it creates overhead for them, it is in their best interest to do so. The hope is that whatever software the user is utilizing, Haystack as an example, will do most of this work for them. It may even be the case that the software is adding things to the repository automatically, and if so it should be easy for the software to identify these two pieces of information. If these parameters are specified, the data dependency layer stores the information by calling the put method of the time layer with the triple (data being added by the user, [Who/Why]Created relation, [Who/Why]Created as supplied by the user). It also, in turn, calls the put method of the time layer with the assertion or bits the user wished to store. The time layer
put method takes in information, and calls the put method of the present triple store with it. It also calls the put method of the present ordinary triple store with the assertion (data being added by user, CreationTime, DateAndTime as supplied by the machine). This, in fact, along with filtering gets when a time interval is specified, is the main function the time layer.

An important circumstance to note here is the case of a buggy service that stores the wrong WhoCreated, thus preventing the user from deleting the bad data by using the data dependency layer. In the case of this happening, the user will simply have to use the ability of the time layer to revert to a state before the buggy service was acting on the Haystack. There is little alternative to this. If the information for WhoCreated and WhyCreated is not passed in, then in must be obtained somewhere else. This would require creating some sort of oracle that knows who is creating data, and why it is being created. The former solution seems simpler, and creates less overhead work for the user.

5.5 Get operations

Get operations can include parameters specifying WhyCreated and/or WhoCreated. In addition, a user can specify a time interval. WhyCreated and WhoCreated are in the form of a URI. A time interval is passed in as two strings. If a specific date and time are being provided, it should be in the TYYMMDDHHMMSS format. This stands for time zone, year, month, day, hour, minute and second, respectively. The reason that the format for a time parameter was chosen to be string is so that the user might alternatively pass in either the string “PRESENT” or “START”. This allows the user to declare the front of their interval to be the beginning of time as seen by their triple store, and likewise to declare the end of the interval to be the present state of the repository.

If the user specifies a WhoCreated/WhyCreated, the results are filtered to only include data that has a correct corresponding (data, WhoCreated/WhyCreated, parameter supplied) assertion also in the repository by the data dependency layer. If
the user specifies a time interval, not only are the results filtered, but the way the query is performed may actually vary. Consider t1 to be the beginning of a time interval and t2 to be the end of a time interval, as declared by the user. If the user specifies both t1 and t2 to be “PRESENT”, then the query will only be performed on the present ordinary triple store. In all other cases, a query will have to be done on both the present and past plain triple stores. The reason for this is that there is no way to determine if the information the user is searching for exists in an undeleted state in the present repository. It could be that some information that existed during the time interval t1-t2 is deleted in the present, as well as it could be that it is not deleted in the present. Since this is the only way that the present and past triple stores are different from each other, both must be searched. However, if the user specifies that they only wish to query the present version, then the past triple store does not need to be accessed. This, in fact, was the main reason for that design decision. Queries on the present are implemented in a much more efficient fashion than queries at points or over intervals of the past. The criteria that must be met for data to be considered part of a declared time interval is simple. All data in the information repository existed at some point in time in a non-deleted state. Specifically, all data exists in this state immediately after being put into the triple store. From that point on, information can be deleted, and subsequently undeleted as many times as the user wishes. This will create intervals of time that the information existed in a non deleted state, as well as intervals that it existed in a deleted state. These intervals will be marked by assertions of the following form: (data, CreationTime, Date and Time), (data, DeletionTime, Date and Time), (data, UnDeletionTime, Date and Time). A pair of the first two assertions, with no other assertions of these forms with date and time between the dates and times in those assertions specifies an interval that the data existed in a non deleted state. If an assertion of the second type exists with no assertion of the third type in existence with a greater date and time, then the data is deleted in the present, has been deleted since the stored date and time, and will be found in the past triple store. This creates an interval that it existed in the deleted state from the stored time up until the present. The same can be said for the third
type of assertion, with the only difference being that it creates an interval that it
existed in a undeleted state up until the present. All other intervals can similarly be
calculated. If an assertion of the second type exists with a date and time before an
assertion of the third type that exists, and there is no other assertion of these types
that exists with a stored date and time between these two, then that data existed in
a deleted state during that interval. Along the same lines, a undeleted interval can
be determined by the existence of an assertion of the third type with a date and time
before an assertion of the second type that is in the repository, and no other assertion
of these type exists for the given data with a time between those two times, then
the data existed in an undeleted state during that time interval. Thus, the criteria
that must be met for data to be considered part of a declared time interval is that
some interval in which it was in a non-deleted state must overlap with the declared
interval.

Finally, if both a WhoCreated/WhyCreated and a time interval are specified, the
results are simply filtered twice. The time layer first filters the results to only include
those which existed in an nondeleted state during the time interval, and passes those
up to the data dependency layer. The data dependency layer, in turn, filters those
results as described above, and passes the final result set back to the user.

5.6 Deletion and Expunge

When one of these operations is performed, information is either removed from the
present triple store and discarded, or added to the past triple store. An exception
is that if a user attempts to delete information that is already deleted. In this case,
nothing will happen. In addition to the information being deleted, any assertions
about the deleted/expunged data are also (re)moved. Examples include Creation-
Time, DeletionTime, UnDeletionTime, WhoCreated and WhyCreated clauses. In
the case of a deletion, a transaction must be started on both the past triple store and
the present triple store. In the case of an expunge, a transaction must only be started
on one of them, depending on which one the data being expunged is currently in.
If a delete or expunge is invoked with a WhoCreated and/or WhyCreated specified, the action is also similar to that of a get. Any data that has a correct corresponding WhoCreated and/or WhyCreated clause associated with it will be expunged or deleted. As before, this would be the same as performing a get operation with the same WhoCreated and/or WhyCreated as parameters, and then deleting or expunging the result set.

If both a time interval and WhoCreated/WhyCreated are passed in to an invocation of the delete or expunge methods, the action is once again similar to that of a get call. All data that existed during the time interval in a non deleted state, and has the correct WhoCreated/WhyCreated clause associated with it will be deleted or expunged. As before, this would be the same action as doing a get with the same parameters and then deleting/expunging the result set.

The aspect of deletion and expunging that separates it from the get operation is propagation. If I delete or expunge a piece of data, any other information that depends on that data, and only that data, should also be taken care of. Let d represent a piece of data that is being expunged or deleted. Any other data that has a clause (data, WhyCreated, d) in the repository and no clause of the form (data, WhyCreated, d') where d is not equal to d', will also be deleted or expunged. This effect then propagates. Any Information that depended on that data, and only that data, in a similar way will be deleted or expunged. It’s important to note that WhyCreated could be a group of objects, in which case an object should only be deleted/expunged as a result of propagation if ALL the objects in that group have also been deleted or expunged.

The final thing that needs to be said about deletion and expunge is in regard to the transaction model. One deletion or expunge, as a result of propagation, could result in many underlying deletes or expunges. Thus, when the user starts a transaction, and performs one of these operations, all resulting deletes or expunges will be considered part of that one transaction. If the user aborts, none of them will take place. Furthermore, if they commit, they are all part of the commit.
5.7 Undeletion

Many of the things that can be said about deletion can also be said about undeletion, but in the reverse direction. When an undeletion operation is performed, a transaction must be started on both the present and past triple stores. Information is being moved from the past triple store into the present triple store, and clauses specifying undeletion times are also being added to the present triple store.

If WhoCreated and/or WhyCreated are specified, then any information in the past triple store that has a (data, WhoCreated, parameter passed in) and/or (data, WhyCreated, parameter passed in) assertion associated with it will be undeleted and moved to the present triple store.

If both a time interval and WhoCreated/WhyCreated are specified, then any data that existed in a deleted state during that time interval, and also has the correct WhoCreated/WhyCreated assertions associated with it will be undeleted and moved from the past triple store into the present triple store.

Undeletion is propagated in the same way that deletion is. If I delete a piece of data, any other information that depends on that data, and only that data, should also be taken care of. Let d represent a piece of data that is being undeleted. Any other data in the past triple store that has a clause (data, WhyCreated, d) in the repository and no clause of the form (data, WhyCreated, d’) where d != d’, will also be undeleted. This effect then propagates. Any information that depended on that data, and only that data, in a similar way will be undeleted. It’s important to note that WhyCreated could be a group of objects, in which case an object should only be undeleted as a result of propagation if ALL the objects in that group have also been undeleted. There is a specific case that needs to be addressed here. Assume that there exists a piece of data Y, and a piece of data X. Y depends on X, and only X. If the user deletes Y, deletes X, and then undeletes X (in that order), Y will reappear. While this is not necessary desirable since the user deliberately deleted Y, it is not harmful so it is considered a acceptable side effect. A possible solution would be to note when information is deleted as a result of propagation, and only consider such
data when propogating undeletion.

A final note about undeletion and is in regard to the transaction model. One undeletion, as a result of propagation, could result in many underlying undeletes. Thus, when the user starts a transaction, and performs an undelete, all resulting undeletes will be considered part of that one transaction. If the user aborts, none of them will take place. Furthermore, if they commit, they are all part of the commit.

5.8 The Evolution of the Implementation

The first idea for how to implement time tracking was to add fields to every straw that stored its creation time, deletion time (if any), and deletion status. While looking into doing this, it was discovered that Haystack already kept track of when each straw was created, in its own tracking data. The idea for tracking data dependency was similar, to add fields to each straw for why it was created and what asked that it be created. Again, it was discovered that this information was being kept track of in Haystack’s tracking data.

The problem with that approach was that it was reliant on existing Haystack code. It seemed more desirable to implement a fancy triple store that could be used by Haystack, but didn’t directly depend on it. Thus, it was decided to make layers for time and data dependency. The only question that created was what those layers were going to rest upon. Obviously, they needed a repository for information. Something that could store a user’s info, as well as the extra information needed to track time and data dependency. The information store that we decided to use was Ken McCracken’s triple store. The reason was simple. He was also working with the Haystack project and the ability to have input into the system that he was designing helped make it better suited to my needs.

As was discussed previously, it was decided to put the time layer directly on top of the plain triple store, and the data dependency layer on top of that. Thus, the fancy triple store was the data dependency layer.
Chapter 6

Conclusions and Future Work

6.1 Future Work

A possibility for future work could be the creation of a unique graphical user interface that allows a user to utilize the features of the work done in this thesis in an efficient and easy manner.

An additional possibility for future work related to this thesis lies in the area that this thesis was born from. The idea of content versioning, while not fully explored in this thesis, is important. If the Haystack system could recognize new documents as new versions of documents already in the repository it could be quite useful to a user. On a basic level, it would prevent duplicate data from existing in the Haystack. More importantly, though, it would make an important association between documents that should be associated. There are several difficult questions that need to be answered before this can be accomplished however. How exactly to determine if a document is a new version of another document is not a simple task. There needs to be a process to determine how similar two documents are, and a way of using that process to decide if two documents are different versions of the same document. Moreover, how to handle metadata related to these documents is not clear. There are certainly some situations where metadata should be shared amongst different versions of the same document, but the question of how much and in which situations would have to be answered.
Another possibility for future work would be to allow a user to actually change information in a past state of their information repository. This raises many issues similar to those of time travel. As is discussed in this thesis, deciding the effects that actions in the past would have on the present is a difficult, if not impossible, task.

6.2 Conclusions

We present a back end to the Haystack system that, when utilized, will allow a much more powerful and comprehensive way for a user to navigate their information. The information repository that was developed allows for the basic task of storing a user’s information, as well as tracking when and why that user stores their information.

Through the tracking of time and data dependence, it allows for a user to feel confident that their system is safe from corruption. They have the ability to revert to any previous state, and to undo the actions of any entity acting autonomously on their information. This is a feature that many users may even find necessary, as Haystack is designed to have many such entities, in the form of services, acting on the information. It also provides for more accountability. If a user notices information entering their repository automatically that they do not desire, it is a simple task to identify the responsible party.

They can also feel more confident that they will be able to locate the information they desire when they desire it as they will have more tools at their disposal for doing so. With the use of this system, they are not bound by the current state of their information. If they desire to search for something that used to be in their information repository, but is not currently, they can search a past state. Furthermore, if they cannot locate something in the current version of their information because it has gotten too cluttered or too disorganized, they can go back to a time when their information was easier to navigate and search there.

We hope that the problems solved by this thesis will improve the Haystack project. The fancy triple store is not currently useful, but can be with future Haystack improvements. Most necessary would be advances for the interface to Haystack. If
adapted, however, a Haystack using the fancy triple store will be a more robust one.
Appendix A

Plain Triple Store Interface

```java
public interface TripleStore {
    /** *
     * These are the values to be stored in the serverPreferences. The first * is
     * the default file from which to read the default preferences if they * are not otherwise
     * specified or obtainable. */
    public static final String preferencesFile = "./serverPreferences"
    public static final String driverNameKey = "TripleStoreDriverName"
    public static final String urlKey = "TripleStoreURL"
    public static final String dataKey = "TripleStoreDataBlock"
    public static final String usernameKey = "TripleStoreUser"
    public static final String passwordKey = "TripleStorePassword"
    /** *
     * Takes care of whatever is necessary to begin interacting with the * store
     * database. * * @exception TsCriticalInitException if there's a problem setting up the
     * TripleStore for use. */
    public void init() throws TsCriticalInitException;
    /** *
     * Begins a transaction in the current thread. * * Make sure that if you call
     * this method you call a commit or abort within * this instance/thread in order to free
     * up the resource. o.w. the * underlying handle will not be freed up for others. */
    public void beginTransaction() throws NoTransactionException
    public TransactionHandle beginTransactionGetHandle() throws NoTransactionEx-
    ```
/** * Attempts to end the current transaction with a commit in the current * thread. Either way, the transaction is closed. */

public void commitTransaction() throws NoTransactionException

public void commitTransaction(TransactionHandle handle) throws NoTransactionException

/** * Attempts to end the current transaction with a rollback in the current * thread. Either way, the transaction is closed. */

public void abortTransaction() throws NoTransactionException

public void abortTransaction(TransactionHandle handle) throws NoTransactionException

/** * Allows the creation of a single rdf statement in Haystack. ** THIS 
METHOD WRITES TO STORAGE ** @param subject the subject of the state-
ment; a URI * @param predicate the predicate; a named relation or URI * @param 
object the receiver of the predicate; a URI * @return the URI of the newly created 
statement * @exception AbortException if transaction should be aborted */

public URI putAssertion(URI subject, URI predicate, URI object) throws Abor-
tException, NoTransactionException

public URI putAssertion(URI subject, URI predicate, URI object, Transaction-
Handle handle) throws AbortException, NoTransactionException

/** * Stores the actual bits for a particular resource, as they resolve in the * current context. ** THIS METHOD WRITES TO STORAGE ** @param URI the 
URI to use to refer to theBits currently being added * @param theBits the serializable 
object whose bits to add to the database * @exception AbortException if transaction 
should be aborted */

public void putBits(URI uri, Serializable theBits) throws AbortException, No-
TransactionException

public void putBits(URI uri, Serializable theBits, TransactionHandle handle) throws 
AbortException, NoTransactionException

/** * Stores the actual bits for a particular resource, as they resolve in the * current context. ** THIS METHOD WRITES TO STORAGE ** @param theBits
the serializable object whose bits to add to the database * @return the URI given to theBits * @exception AbortException if transaction should be aborted */

    public URI putBits(Serializable theBits) throws AbortException, NoTransactionException

    public URI putBits(Serializable theBits, TransactionHandle handle) throws AbortException, NoTransactionException

    /** * Gets the associated subject, predicate, or object for the rdf statement * indicated. * * THIS METHOD DOES NOT WRITE TO STORAGE * * @param rdfStatement the uri of the associated statement * @exception AbortException if transaction should be aborted * @return the URI of the sub/pred/obj, or null if no such statement */

    public URI getSubject(URI rdfStatement) throws AbortException, NoTransactionException

    public URI getPredicate(URI rdfStatement) throws AbortException, NoTransactionException

    public URI getObject(URI rdfStatement) throws AbortException, NoTransactionException

    public URI getSubject(URI rdfStatement, TransactionHandle handle) throws AbortException, NoTransactionException

    public URI getPredicate(URI rdfStatement, TransactionHandle handle) throws AbortException, NoTransactionException

    public URI getObject(URI rdfStatement, TransactionHandle handle) throws AbortException, NoTransactionException

    /** * Gets the associated serializable object from the bits related to the * given identifier. * * THIS METHOD DOES NOT WRITE TO STORAGE * * @param bitsKey the identifier for the resource desired * @exception AbortException if transaction should be aborted * @return the bits if bitsKey has a value, null otherwise */

    public Serializable getBits(URI bitsKey) throws AbortException, NoTransactionException
public Serializable getBits(URI bitsKey, TransactionHandle handle) throws AbortException, NoTransactionException

/** * Gets a list of all assertions with the particular URI as subject/object/ * etc. * * THIS METHOD DOES NOT WRITE TO STORAGE * * @return the appropriate Vector[URI].elements(), which may be empty * @exception AbortException if a problem with read */

public Enumeration getAssertionsBySubject(URI subject) throws AbortException, NoTransactionException

public Enumeration getAssertionsByPredicate(URI predicate) throws AbortException, NoTransactionException

public Enumeration getAssertionsByObject(URI object) throws AbortException, NoTransactionException

public Enumeration getAssertionsBySubject(URI subject, TransactionHandle handle) throws AbortException, NoTransactionException

public Enumeration getAssertionsByPredicate(URI predicate, TransactionHandle handle) throws AbortException, NoTransactionException

public Enumeration getAssertionsByObject(URI object, TransactionHandle handle) throws AbortException, NoTransactionException

/** * Gets a list of all assertions. * * THIS METHOD DOES NOT WRITE TO STORAGE * * @return the appropriate Enumeration[URI] of all assertions * @exception AbortException if a problem with read */

public Enumeration getAllAssertions() throws AbortException, NoTransactionException

public Enumeration getAllAssertions(TransactionHandle handle) throws AbortException, NoTransactionException

/** * Returns the next available new, unused URI from the database. * * THIS METHOD WRITES TO STORAGE * * @exception AbortException if a problem getting next URI */

public URI nextURI() throws AbortException, NoTransactionException

public URI nextURI(TransactionHandle handle) throws AbortException, NoTrans-
actionException

/** * Tests whether the specified uri is present in the haystack. If the uri * has
been deleted or expunged, this method will return false. ** THIS METHOD WILL
NOT WRITE TO STORAGE (for now it does nothing) ** @param uri the id to
look for in haystack ** @exception AbortException if transaction should be aborted
*/

public boolean containsURI(URI uri) throws AbortException, NoTransactionException

public boolean containsURI(URI uri, TransactionHandle handle) throws AbortException, NoTransactionException

/** * "Expunges" the given uri and anything else that this uri directly affects
* (e.g. all rdf statements with this uri as object). Expunging means * removal of
the associated bits from disk and cannot be undone. ** THIS METHOD WILL
WRITE TO STORAGE (for now it does nothing) ** QUESTION: SHOULD THIS
METHOD, OR ANY FORM OF DELETE/REMOVAL, BE AVAILABLE? ** @param
uri the uri to be removed ** @exception AbortException if transaction should be
aborted */

public void expunge(URI uri) throws AbortException, NoTransactionException

public void expunge(URI uri, TransactionHandle handle) throws AbortException, NoTransactionException

/** * Terminates connectivity with storage medium with given instance. To *
reinitialize storage interaction, it is necessary to instantiate a new * TripleStore in-
stance. */ public void close()
Bibliography


