Integrating Structural Search Capabilities into Project Haystack.

1. Introduction.

As more and more aspects of one's life have moved online, the amount of digital information the user accumulates and processes has grown tremendously. From personal emails, to electronic receipts from an online shopping site, to messages containing one's ariline reservations - the data is becoming increasingly difficult to keep track of. The Haystack project is an attempt to create a personal information retrieval system that would help the user search through and organize his body of knowledge.

The kind of search one can perform depends on what aspects of the document get categorized and indexed. On one end of the spectrum, there is unstructured (textual) search, where the system indexes the documents by their language features. At the heart of such a search engine is an inverted file index which maps words to the documents that they occur in. On the other end of the spectrum there is a structural (database) search. Structural search often deals with the document's metadata (its date, title, author etc) which is organized into a predefined table. Since the metadata is often numeric, structural searches can be used to answer queries that require comparisons between the values stored in the table, like for example finding "all documents written after 03/01/99."

Haystack is not "yet another search engine." Rather, it's designed to be an interface to various types of information retrieval systems. When a user types in a query, haystack should analyze it, dispatch appropriate requests to the underlying IR systems, and afterwards combine the results. For example, if the user requests "all documents about California that I've read in the last week," haystack would dispatch the task of finding all documents about California in the user's repository to a text search engine. The second part of the query is a structured query and it asks haystack to limit the result set to a certain time period. This task would be dispatched to the structural (database) search engine.

Currently, Haystack incorporates only one information retrieval system and performs only unstructured search. The goal of this thesis is to design and implement structural search capabilities for Haystack. In particular, this implies designing a scheme to store the metadata about the documents, adding the appropriate user interface features, integrating a relational database, and formulating a way to translate user requests into appropriate database queries and interpret the results.

2. The Data Model.

The current Haystack data model is essentially a labeled directed graph strucutre, where nodes and edges are first-class objects, and each node has a unique ID. Nodes in

Haystack are called Straws. Straws subdivide into three subtypes: needles, ties, and bales. A needle is a straw that contains a piece of "raw data". It is essentially a wrapper around some Java data structure (a file, an array, a String etc.) A tie represents a directed relationship between two straws. For example, if a straw A represents a document and a straw B represents a person who wrote that document, than the relationship could be expressed via the "Author" tie. Most of the straws and ties are created internally by haystack, but the user can create a tie between two documents if he wants to make an association between them.

A bale is a centerpiece that represents an aggregation of straws and the relationships between them. A bale can be used to represent a document, a person, or a query. For example, For example, if we have a postcript document called Design.ps, created on 12/02/99, it will be represented in the data model as follows:

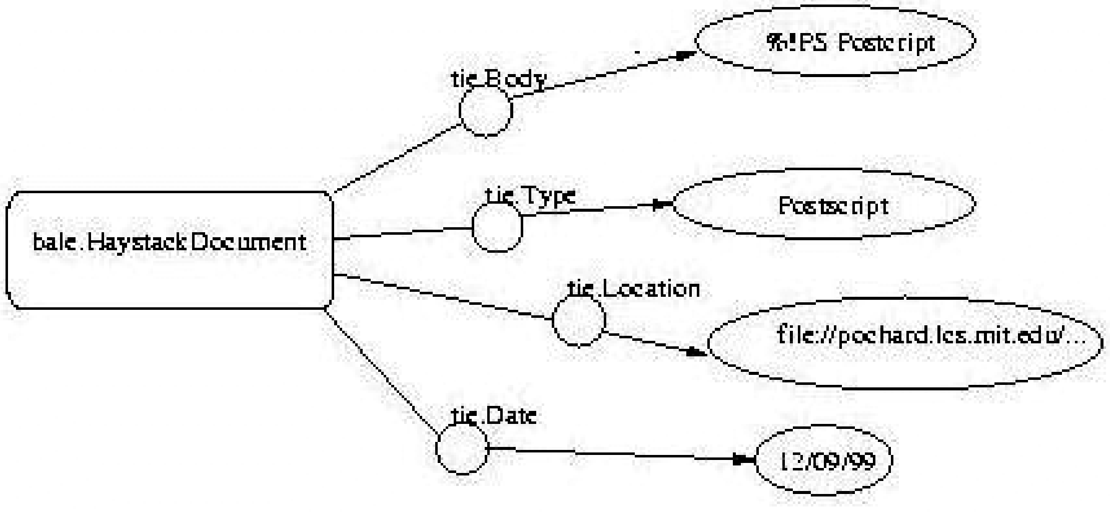

Figure 1. Representation of a Haystack Document

This model does not easily lend itself to be transformed into a table for structural searches. Even though it's quite likey that most documents will have certain attributes (e.g. Date) associated with them, this fact is not at all guaranteed to be true (aside from tracking and debugging information maintained internally by Haystack for each document, which should be of no interest to the user.) Moreover, different types of documents will have completely different attributes - an HTML file will have "links to" attribute, while an email message will have "from", "to", and "subject" fields. Also, since the user is allowed to create links between documents , we can't even predict what type of relationship he will come up with.

One relationship, however, is guaranteed to exist in our data model. Every edge is guaranteed to have a "from" and a "to" node. Note that the edges are typed (e.g. in Fig. 2, the link from the root node to the Author node is labeled Tie.Author), and typing is the key feature that will allow us to infer information about the structure of our data. So in order to perform structural queries, we are going to represent the data graph in 2 tables: the first one (we'll call it TiesTable) will correspond to the relations between needles based on tie types, and the second one (called NeedlesTable) will actually store the data contained in those needles.

An interesting question to resolve is whether or not we should somehow make use of the type of a needle (which in essense represents the type of the data stored in the needle.) While theoretically, we can put all the data into one column of the NeedlesTable, practically, we may have to specify the type of data that's stored in that column when creating the NeedlesTable. So we may end up having to create a separate NeedlesTable for every type of needle that we have in haystack, and have an additional table map needle IDs to their types. Unlike the tie types, needle types cannot be user-defined, so the problem of "unpredictable" types will not arise.

In principle, we could maintian the invariant that every HaystackID that appears in FromID or ToID column of TieTable is a primary key in the second table, but since ties can connect not only needles but other ties as well, this would imply that certain entries in the second table will contain no data. Instead, we can just conclude that if an ID is not found in the second table, that means it's not a needle and contains no data.

TieID	**FromID**	**ToID**	**Label**
4567	4566	4568	Tie.Author
5432	1456	5734	Tie.Date
5867	1456	1235	Tie.DocType
2345	4742	1245	Tie.DocType
2346	1456	1258	Tie.Subject
8766	9876	1245	Tie.DocType
8767	9876	1258	Tie.Subject
...	...	...	...

Fig.2 TiesTable

ID	Data
4568	"John Smith"
5734	"03.01.99"
1235	HayMIMEData.Email
1245	HayMIMEData.Postscript
1258	"Your monthly phone bill statement"
...	...

Fig. 3 NeedlesTable

An alternative way to represent the data could be for example making a separate table for each type of Tie. Though this may be a conceptually simpler way of looking at the data, it does not answer the question of what to do with "unpredictable" user-defined types of ties. Furthermore, our model already incorporates this model - we can obtain a table for a particular type of tie using the view command (explained below.) Note also that in our model, TieID is a good choice for a primary key since TieIDs are unique in our data model and hence in our TiesTable, the primary key is never repeated.

3. The Queries.

Now that we have defined the schema for the databases, we'll illustrate how the user requests can be transformed into database queries. We'll write the queries in SQL (Structured Query Language), which is the standard language for relational database management systems. See http://w3.one.net/~jhoffman/sqltut.htm for SQL tutorial.

Suppose the user wants to find to see all emails messages with the subject "Your monthly phone bill statement". So what we need to do is find all the needles that have a Tie.DocType leading to the needle containing "HayMIMEData.Email" AND a Tie.Subject leading to the needle containing the desired string.

The corresponding database query will look as follows:

First we select all needles that that satisfy the type condition.

```
CREATE VIEW EMAILS AS

   SELECT TiesTable.FromID FROM TiesTable, NeedlesTable

   WHERE TiesTable.Label = 'Tie.DocType' AND
                TiesTable.ToID = NeedlesTable.ID AND
                NeedlesTable.Data = 'HayMIMEData.Email'
```

Then we select all needles that satisfy the subject condition:

```
CREATE VIEW PHONEBILLS AS

   SELECT TiesTable.FromID FROM TiesTable, NeedlesTable
   WHERE TiesTable.Label = 'Tie.Subject' AND
                 TiesTable.ToID = NeedlesTable.ID AND
                 NeedlesTable.Data = 'Your monthly phone bill statement '
```

Finally, we perfrom a join and return only those needles that are in both views.

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

For our example table in Fig.2, this will return {1456, 9876}.

Let's consider another example. Suppose you want to see if any of the documents in your haystack mention the paper that you wrote two years ago (entitled "MyPaper" for simplicity) in their bibliography. In order to answer this query, we would first need to locate the bale corresponding to the paper you have in mind. Then, we need to search our TiesTable to see if any of the relations are labeled Tie.References and point to the ID of the bale for your paper.

Similarly to the previous example, we can create the views WRITTEN_2YRS_AGO and TITLE_MYPAPER. Then, the query for the references will look as follows:

```
SELECT FromID FROM TiesTable
WHERE Label = 'Tie.References' AND
             ToID IN

                 (SELECT FromID FROM WRITTEN_2YSR_AGO, TITLE_MYPAPER
                 WHERE WRITTEN_2YRS_AGO.FromID TITLE_MYPAPER.FromID)
```

This exampe raises an interesting issue: when a user makes a query looking for all documents that reference the one named "MyPaper" it could be the case that "MyPaper" has been archived and is a separate bale that we should be looking for (as in example above.) However, it's also possible that Tie.References points to a needle containing just the name of the document, in which case our previous query to the database will not return the correct results. The solution to this problem seems to be to either let the user specify whether the document he has in mind has been archived, or to have haystack perform both queries and combine the results.

4. Persistent Storage and Updates to the Database.

Up to this point in the discussion, we have assumed that the relational database already contains the data in the user's Haystack. Now we'll attempt to describe a mechanism to guarantee this assumption.

Currently, the haystack data graph is stored in a persistent non-relational database. The data structure used is a hashtable, where the keys are HaystackIDs and the values are the straws that correspond to those IDs. The database supports transaction-protected operations. That is, if a given service modifies the data graph, then all of its modifications are grouped into a single transaction, and either all of them take place (if the transaction commits) or none of them do (if the transaction is aborted.)

Ideally, in order to implement structural search capabilities, it would make sense to switch to a relational database as a means of persistent storage. However, as it seems to be hard to find a non-commercial relational database that supports transactions, we have to stick with having 2 databases - one that supports transactions and is used for persistent storage, and another (relational) one that has a copy of the data and can be used to perform structured queries.

In order to maintain a copy of the data in the relational database, we we need to devise a mechanism to:

a) build the relational database from the data stored in the persistent database,

b) update the relational database accordingly as the user archives new documents into his haystack.

Even though we know we can maintain the relational database by rebuilding it from the transactional one every so often, a more efficient solution seems to be to actually propagate the incremental changes to the relational database, and rebuild it from the transactional one only if we have a reason to believe that the data in the relational database is inconsistent or corrupted.

In doing the incremental updates, we need to make sure that TiesTable and NeedlesTable are consistent with each other (i.e. it should not be the case that TiesTable contains a Tie.DocType link from ID 1456 to ID 1235, but the needle with ID 1235 is not in the NeedlesTable.) We should also make sure that all of the service's updates happen atomically and if a service is aborted, then none of its updates take place in the database.

One way of implementing this would be to store all the update requests a service makes in a persistent queue (just like we do with the Service queue). This batch of requests will be visible on the queue only if the service commits. In attemption to ensure that all requests happen atomically, we could mark each individual update request as it's being executed, and then if all are marked - the batch is removed from the queue. If upon restarting haystack we notice that a particular batch of update requests contains marked elements - then we can either undo the changes, or carry out the rest of the updates. However, we should also keep in mind that perhaps in the future, we may get access to a relational database that supports transactions, so our design should not be dependent the above implementation details, so that a relational database with transactional support could be easily plugged in.

5. Accessing the database from the Java code

We would like to make our design modular enough so that it could work with any relational database that supports the set of features we need. The JDBC API appears to be the ideal solution for this problem. The JDBC API allows an application to access virtually any database on any platfrom with a Java VM, as long as the driver for that database implements the JDBC API.

Working with JDBC API usually involves 3 basic steps: connecting to the database, creating and executing a statement, and processing the result set. For example, it may look as follows:

```
 Connection con = DriverManager.getConnection("jdbc:myDriver:wombat",

                                   "myLogin", "myPassword");

   Statement stmt = con.createStatement();

   ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");

   while (rs.next()) {

         int x = rs.getInt("a");

         String s = rs.getString("b");

         float f = rs.getFloat("c");
```

As one can notice, aside from connection, there is virtually no dependence on the database itself, and majority of the code involves working with JDBC Statement and ResultSet classes.

6. User Interface

Ideally, once the structured query capabilities are implmented, one could develop an interface that recognizes natural language queries and dispatches the appropriate requests to the database and the text search engine, and then combines the results.

That, however, is a goal for the future. In the meantime, we would concentrate on providing a structured query-specific user interface. In designing this interface, we want to give the users the maximum flexibility in specifying what attributes to search for and what conditions to impose on those attributes. Roughly, the user interface will look as follows: For every attribute, it will consist of a text box in which the user can type the name of the attribute. The text box will also contain a drop-down list of all attributes (ties) currently known to haystack, so that the user does have to remember exactly how we name a relation in haystack.

The attribute field will be followed by the box where a user can choose the type of constraint on the value of the attribute (basically corresponding to the conditional SELECT operation) - whether in should be IN a certain set, BETWEEN two values, LIKE (match) a certain expression, EQUAL a certain value and so on. We would like to further constraint the user's choices based on the attribute that he specified (for example, it's highly unlikely that the name attribute will be anything other than a string, and it doesn't make sense (except if you impose lexicographical order) for a string to be greater than a certain value.) This point, however, may need further consideration since currently in Haystack , the type of the Tie does not impose any conditions on the type of the needle it points to.

Based on the type of constraint, the number of additional input fields will be adjusted (e.g. if it's BETWEEN, then we need 2 values, if it's IN, then we need at least one more input field, plus a button that will generate more fields if needed.)

There will also be an AND and OR buttons, which will prompt the appearance of additional input fields for the next attribute.

Before the graphical interface is put in place, we will also need to come up with a command-line interface to use at least for debugging purposes. The command-line interface will most likely consist of the main squery command, plus two auxiliary functions: listTies and listConstraints (which are self-explanatory). If the specified conditions cannot be applied to the given types of attributes, the command will return an error.

7. Project Milestones.

The first step in developing structural search capabilities would be to integrate a relational database into the Haystack code. This will involve downloading and installing the database (most likely mySQL), writing the the code that connects to the database via the JDBC API, and then creating/accessing sample databases.

Once the database is in place, we would proceed to create the two databases described in the design and implement the part that populates the relational database with Haystack data (perhaps without queuing up the requests at first.) The expectation is to have this working by the end of IAP.

The next step would involve writing a structural query service that would transform user requests into SQL database queries and process the results returned by those queries. This seems to be a more intersting part of the project. In oder to test the service, we will also need to write a command-line interface described above. Tentatively, it should be completed in February/first half of March.

The final step would be attaching (probably Web) graphical user interface outlined above, which, if everything goes smoothly, should be in place by the end of March/first half of April, leaving over a month for the final thesis writeup.

8. References.

1. Eytan Adar. Hybrid-Search and Storage of Semi-Structured Information. MEng Thesis. May, 1998

2. Ilya Lisanskiy. A Data Model for the Haystack Document Management System. MEng Thesis. February 1999.

3. JDBC Technology Guide: Getting Started. http://java.sun.com/products/jdk/1.3/docs/guide/jdbc/index.html

4. Introduction to Structured Query Language. http://w3.one.net/~jhoffman/sqltut.htm