Flexible Information Retrieval

Harvesting the ramblings of an idle blogger

Carlos Torchia (V00717432). Supervisor: Alex Thomo.
CSc 499: Technical project. Spring 2011. University of Victoria.

Introduction

The web is full of unstructured data. In particular, many idle bloggers ramble in their personal blogs, and the little interest in these ramblings may cause them to be overlooked in scientific inquiry. *Thinklog* is a program that wants to make such ramblings useful.

What kind of new information we can infer when we have access to the thoughts of a blogger? As our lives and our minds are cluttered with information overload and preoccupation of survival, we need to listen more attentively to our thoughts. It is hoped that an application like this, which presents inferred data about users' thoughts, can help in bringing to the surface thoughts which would otherwise be silenced by the wants and worries of a typical human being.

The goal of this project is to design and implement the Thinklog system, and this paper will discuss its design and implementation. I have coded the features described in this paper during the course of a semester.

To make thoughts useful, the thoughts can be stored as text, and information about them can be stored in and retrieved from a knowledgebase in the form of a semantic network using the Resource Description Framework (RDF). This paper will discuss how this is done. We discuss:

1. Other projects attempting to store knowledge in RDF
2. Goals of this project
3. Implementation of the application
4. Optimization of the application
5. Conclusions

In this paper, the word *knowledgebase* is used many times in the same way as "database", and in this case could be thought of in that way.

Other projects

Other people have played with the idea of inferring knowledge from posts that people write. A project like this is Twarql [1], which streams tweets from Twitter [4] and figures out business information and sentiment from them, storing the inferred info in a semantic network so that queries can be performed on them using a user-friendly query buildter. Thus, this project takes unstructured data that is on the web, and organizes them into an RDF graph.

An even bigger and more general project is DBPedia, which extracts facts like country capitals and concept subsumptions from Wikipedia articles and stores them in an RDF triplestore [2]. Thanks to this project, the large amount of data on Wikipedia is stored in a way so that applications can easily query this data, that is, using an RDF graph.

Like these projects, I use a semantic network in an RDF triplestore to represent a knowledgebase. RDF lets us represent knowledge in a way that allows us to easily update our schema, and it lets us publish knowledge on the web or retrieve knowledge from other sources using an open standard.

Goal: a thought knowledgebase

Here I will discuss the goal of this project. The goal is to design and implement a system that queries and infers from information about thoughts in a knowledgebase. To accomplish this task, we need to focus energy on **storing**, **maintaining**, and **updating** the knowledgebase.

The knowledgebase is to be in the form of a store of RDF triples, or a *triplestore*. The system could be asked to fetch blog posts from the web or from a user input form, and then it will infer knowledge from the text of these thoughts. When the triples representing this knowledge are in the database, they can be used to answer queries and make inferences. Inferences may be made using information from other applications on the web, like Wikipedia. Pulling in info from other sites would make this app a *mashup* between the thoughts on Twitter and other social networks, and Wikipedia.

Thus, the function of Thinklog shall include:

1. Fetching thought posts from social networks like Twitter,
2. Extracting knowledge from thought text,
3. Making queries for thoughts based on relevance to query,
4. Making inferences in the knowledgebase to help in producing info / search results for user,
5. Updating the knowledgebase upon arrival of new thoughts,
6. Using information from other web apps to make inferences,
7. Publishing thoughts, displaying knowledge in a useful way, and
8. User input of thoughts

Now implementation of the Thinklog system will be discussed.

Implementation: a keyword knowledgebase

Here, I discuss the specific way in which I coded this application, including storing the data, inferring knowledge, and the user interface. Although I wanted a system that could infer very general knowledge from thought texts, the main kind of knowledge the system I coded can infer is that of which **keywords** are relevant to the thoughts in the knowledgebase. The following list contains the abilities of the system which I implemented with respect to each of the above goals.

1. Fetching thought posts from RSS feeds in social networks like Twitter,
2. Determining relevant keywords of thoughts from their text,
3. Making queries for thoughts based on relevant keywords in common with a query or a certain thought,

4. Inferring the relationships between each of the keywords of all the thoughts, and inferring the total number of occurrences of each keyword, to help in determining the importance and relevance of keywords,

5. Updating the keyword relationships in the background

6. Using Wikipedia's API to determine to infer what keywords are common, even if they are uncommon in the knowledgebase,

7. Publishing thoughts in the web application,

8. Generating a tag cloud, i.e. a list of common keywords, for each thought, and

9. User input of thoughts

The details of the implementation of these features follows.

Knowledgebase implementation: triplestore

As I mentioned at earlier, the knowledgebase used to represent information about the thoughts is in the form of a *semantic network*. A semantic network can be visualized as a graph, with the nodes representing individuals or concepts in our knowledgebase, and with edges representing the relationships between the nodes. Here, subject and object nodes are related with predicates. This way, you can represent propositions like "dog chases cat", or "peace is related to happiness". The semantic network is implemented using an RDF triplestore.

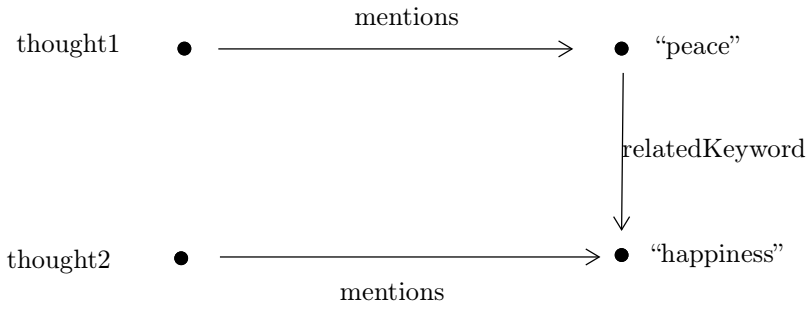


Figure 1 This graph represents relationship btwn. thoughts and keywords.

Each subject-predicate-object relationship between two nodes, called a *triple*, of the semantic network are stored in the database as a *triplestore*. Here, we have a set of triples in the form of (subject, predicate, verb), which describe the various propositions of this knowledgebase. In Thinklog, most publicly accessible data, including thought text and keyword info, i.e. everything except for passwords, is stored in this triplestore. The following main relationships are maintained by the application in the triplestore:

- *mentions* Thought T mentions keyword W if W occurs in the text of T.

- *relatedKeyword* Keyword W_1 is related to keyword W_2 if W_1 and W_2 occur in sufficiently many thoughts together.

- *commonKeyword* Keyword W is a common keyword if it occurs in sufficiently many thoughts.

- *count* Keyword W has a count C if W occurs in C thoughts; or, keyword pair W_1, W_2 has count C if W_1 and W_2 occur in C thoughts together.

- *content* Thought T has content B if B is the content (text) of T

- *author* Thought T has author A if the user who submitted T was logged in as A

- *date* Thought T has date D (UTC seconds since epoch) if it was written at that date

The nodes in the graph consist of thoughts, thinkers, and keywords. Other relationships exist to identify thinkers and thoughts with IDs and names, etc.

Resource Description Framework

To implement the triple store for my code to use, I used the ARC 2 library for the Resource Description Framework (RDF). This library maintains a triple store in a MySQL database with triples, subjects, objects, and predicates stored in their own tables and normalized by ID. It maintains B-tree indexes for selecting triples by subject-predicate, predicate-object, and subject-object pairs. ARC 2 also provides a query language called SPARQL+, which allows specifying queries and insertions in a way similar to SQL [3]. This allows the system to query for knowledge represented as triples in the triple store, and to insert these triples when thought info needs to be added or updated. Examples of SPARQL queries used in this project are in later sections of this paper.

The choice of using RDF and semantic networks to store thought information is due to its flexibility with regard to the range of data we can store and the ease of schema changes. All the thought data of this application (aside from username and passwords) is stored in one table, which is the triple store. The relationship between entities is described in each record. This allows us to bring in new relationships without having to alter the database schema.

Currently, most of the knowledge that is inferred from user input thoughts has to do with keywords of each thought, and keyword interrelationships. However, it may be that **more** things about the thoughts can be inferred. The flexibility of RDF allows us to easily add these relationships. It could be that relationships themselves may be specified by the user, who doesn't have access to the database schema. This will allow us to flexibly obtain a more useful knowledge-base.

Issues with the schema

The current scheme is prone to not distinguishing between the relevance of some keywords to a thought and the relevance of different keywords to the same thought. This is because the keyword relation does not capture the **number** of occurrences of the keyword in a particular thought, nor the number of thoughts it occurs in. The keyword threshold mitigates this by keeping seldomly used keywords from contributing the the relationship, but once it passes that threshold, any article that has that keyword is related to it.

However, the thoughts are short, and are not likely to have repeated keywords, so multiple keywords appearing in a thought implies that the keywords are probably related. For example, a thought containing "anxiety" and "blue crayons" could only be something short like "having anxiety is like having only blue crayons", because keywords in a short thought have to be close together, which means they're probably related.

Issues with the database system

It is possible that by having a large triplestore, which I am assuming is stored in one large table, the size of the triplestore may exceed the maximum table size in MySQL.

Updating the triplestore for new data can involve more complicated queries than we would have using a fixed set of tables to represent keyword relationships. For example, just to update the number of occurrences of a keyword in the set of all thoughts, we need to first find if the triple for the count is in the triplestore, delete the count triple relating this keyword to this count, and then we need to add a new triple indicating the updated count. This could be done with a single simple `UPDATE` statement in SQL.

Tagging: assigning keywords to thoughts

In order to query for thoughts relevant to a set of words given by the user, it is necessary to tag each thought with a set of *keywords* that pertain to that thought. The problem is to determine

1. Which words are important enough to be keywords of a thought
2. What makes a thought's keyword relevant to a given query

To do these things, we store triples in the triplestore that represent the facts that certain keywords are important and that certain keywords are related to each other. Updating this information is done by a background process, and this information is inferred from the number of occurrences of these keywords. Then we can quickly find which words of a thought are keywords, and use that information to return search results for a query.

Determining the keywords of a thought

When we want to add a thought to the system, we must consider which words are important enough to be keywords of the thought. To do this, when looking at the thought's text, we determine if any of the following hold:

1. A word starts with a "#" character (such a keyword is called a *hashtag*)
2. A word already is flagged as a common keyword
3. A word is in Wikipedia

We proceed to discuss these three conditions in detail.

Hashtags

Hashtags are necessary so that if a user wants to tag the thought with a keyword, but has no other guarantee of this happening, then he or she can do so with simply prepending "#" to the word. For example, if I want a keyword of my thought to be "computer", I simply prepend "`#`" and put "`#computer`" where it occurs in the sentence, so we have sentences like "`my #computer is a #netbook`". This is done by users of Twitter also.

If a keyword consists of multiple words, a user must hashtag it using underscores ("_") for the spaces. For example, "computer science" would be hashtagged as "`#computer_science`". Then, when "computer science" appears in a sentence somewhere, it is matched to the keyword referred to by this hashtag, and will appear relevant to other thoughts with this phrase.

Common keywords

When a thought has been added to the system, its words are collected, and Thinklog checks to see if each word is a keyword of the thought. If many other thoughts already mention the keyword, then we can assume the word is important, and we tag the thought with that keyword. The number of *mentions*, i.e. the number of thoughts that have mentioned a word, that is necessary to make a plain word flagged as a keyword is called the *keyword threshold*.

For example, if "anxiety" is mentioned in at least 5 thoughts, it passes the threshold, and "anxiety" is then considered an important keyword, and other thoughts that contain the word but do not have it hashtagged will mention it. Note that the first 5 thoughts mentioning "anxiety" have to have it hashtagged as "#anxiety".

Currently, the keyword threshold is set to a value of "5". The keyword threshold is set so low for testing purposes, and probably should be higher, so we are left without doubt that the keyword is important when it has passed the keyword threshold.

Wikipedia keywords

Presently, a routine exists in the updating process to scan the set of thoughts for all possible words, and then query Wikipedia's API to see if there is an article in Wikipedia for the word. If it does, it inserts a triple indicating that the keyword is a *CommonKeyword*. This is very slow (see Optimization).

Related keyword pairs

If two keywords occur in the same thought, and many other thoughts also have those keywords together, then we can assume they are related, and we update our knowledgebase to reflect this fact under the *relatedKeyword* predicate. The minimum number of thoughts in which two words need to occur together in in order to make them related is also set to the keyword threshold.

For example, if the words "algorithm" and "panic" occur together in at least 5 thoughts, we add the triple ("algorithm", relatedKeyword, "panic") and its converse.

Determining relevance

When we want to query for thoughts that are relevant to a set of words, we must consider what makes a thought relevant to the query. To find such thoughts, we scan the knowledgebase for every thought satisfying the following conditions:

1. The thought mentions some of the same words as the query
2. The thought mentions a word that is related to a word in the query (see schema above)

The resulting thoughts are then sorted in descending order of the number of words they have in common with the query, and then by the number of words they have that are related to words in the query, and then by date. This way, the user is given the most relevant and recent thoughts first. The following is a simplified example of the SPARQL query used to do this:

```
SELECT ?thought, COUNT(?keyword) AS ?count1, COUNT(?keyword2) AS ?count2
WHERE {
  ?myThought mentions ?keyword.
  { ?thought mentions ?keyword. }
  UNION
  { ?thought mentions ?keyword2.
    ?keyword2 relatedKeyword ?keyword. }
```

```
    ?thought date ?date.

} GROUP BY ?thought ORDER BY DESC(?count1) DESC(?count2) DESC(?date)
```

This SPARQL query is given to the ARC 2 procedure that translates the query into SQL, and then queries the MySQL database for those triples, binding the requested values from the triples to the output fields. The output is a series of rows with thought and keyword count that the application uses to return search results.

If we want thoughts that are related to a thought that the user is viewing, which are listed in the "Related Thoughts" section in the web app, we treat the query as the set of words that the thought mentions. Thus, we look for thoughts that have words in common with or related to words in the specified thought.

The user can also make a query for all thoughts, and then the system returns all thoughts in order of descending date.

Updating thought relationships

Currently, the tagging system updates the keyword and thought triples in a background process called `updater`. The system also looks for keywords for each thought when it is being added by the user, updating the thought-keyword-mentions. It does not update the keyword frequency information, however.

Issues with the tagging system

Above I discussed the issues with the schema for maintaining keyword relationships, which was that the system does not provide information about the number of occurrences of a keyword in each thought.

Another problem with the way Thinklog tags keywords in the thought text is that it does not take into account the full grammar of the English language. For example, "computer" and "computers" would currently be considered different concepts by the system. More than you would think, this is tricky to resolve, because many words that are not plural end with "s" (like "Hercules", or "gallows"). Furthermore, Thinklog may not tag a keyword correctly because of the ambiguity of language. For example, a thought talking about a "lemon" may have nothing to do with the citrus fruit, but rather about kind of car.

Returning useful information

Tag cloud

Users could find it useful to receive information based on the overall content of their thoughts. The most basic functionality that provides this information is a *tag cloud*, which is a set of keywords that pertains to what the user is looking at on the screen. In Thinklog, a tag cloud is displayed that contains the keywords mentioned by the thought(s) being displayed or which the thinker whose thoughts are being searched. It also returns keyword pairs that are related by the `relatedKeyword` relation, and are mentioned by the thought(s). It prints both keywords and keyword pairs in order of decreasing number of thoughts mentioning both.

Keywords and keyword pairs are found by doing a SPARQL query for the keywords that occur in the greatest number of thoughts, and the keyword pairs that occur in the greatest number of thoughts together. The following is a simplified SPARQL query for getting a tag cloud for the most common keywords for the given thinker:

```
SELECT ?keyword WHERE {

  ?thought mentions ?keyword.
```

```
    ?thought author ‘‘{CURRENT_THINKER}’’.

    ?keyword count ?cnt.

} ORDER BY DESC(?cnt) LIMIT 5
```

As you can see, we use the *count* relation to get the top 5 most frequently occurring keywords. The other way to do this would be to use a `GROUP BY` statement, grouping by `?keyword` and counting the number of thoughts that have it:

```
SELECT ?keyword COUNT(?thought) AS ?cnt WHERE {

  ?thought mentions ?keyword.

  ?thought author ‘‘{CURRENT_THINKER}’’.

} GROUP BY ?keyword ORDER BY DESC(?cnt) LIMIT 5
```

See the Optimization section below for why this is not done.

To get the tag cloud for the thought presently being displayed, we don't care about the keyword counts because there will only be a few keywords anyway:

```
SELECT ?keyword WHERE {

  ?thought thoughtId ‘‘{CURRENT_ID}’’.

  ?thought mentions ?keyword.

}
```

These queries are similar for finding keyword-pair relationships. Here is the query for finding keyword relationships for the logged in user:

```
SELECT ?keyword1 ?keyword2 WHERE {

  ?thought mentions ?keyword1.

  ?thought mentions ?keyword2.

  ?thought author ‘‘{CURRENT_THINKER}’’.

  ?rel keyword1 ?keyword1.

  ?rel keyword2 ?keyword2.

  ?rel count ?cnt.

} ORDER BY DESC(?cnt) LIMIT 5
```

For this, we store blank nodes that point to a keyword pair `(keyword1,keyword2)` and a count for the pair, to record how many thoughts the pair of keywords occurs in together.

Other ideas

Another idea would be to take the keywords in the tag cloud associated with this thought, and present images from Flickr or other image hosting web sites that appear in the results of a search using those keywords. This would give the user visual feedback on their typed thoughts.

User interface: uploading thoughts

The user can input thoughts into Thinklog via the standard way: typing them in a text box and hitting the "submit" button. However, the user can also import thoughts described in RSS feeds. Since many sites provide RSS feeds, this allows a wide range of users to upload thoughts.

Uploading thoughts from Twitter

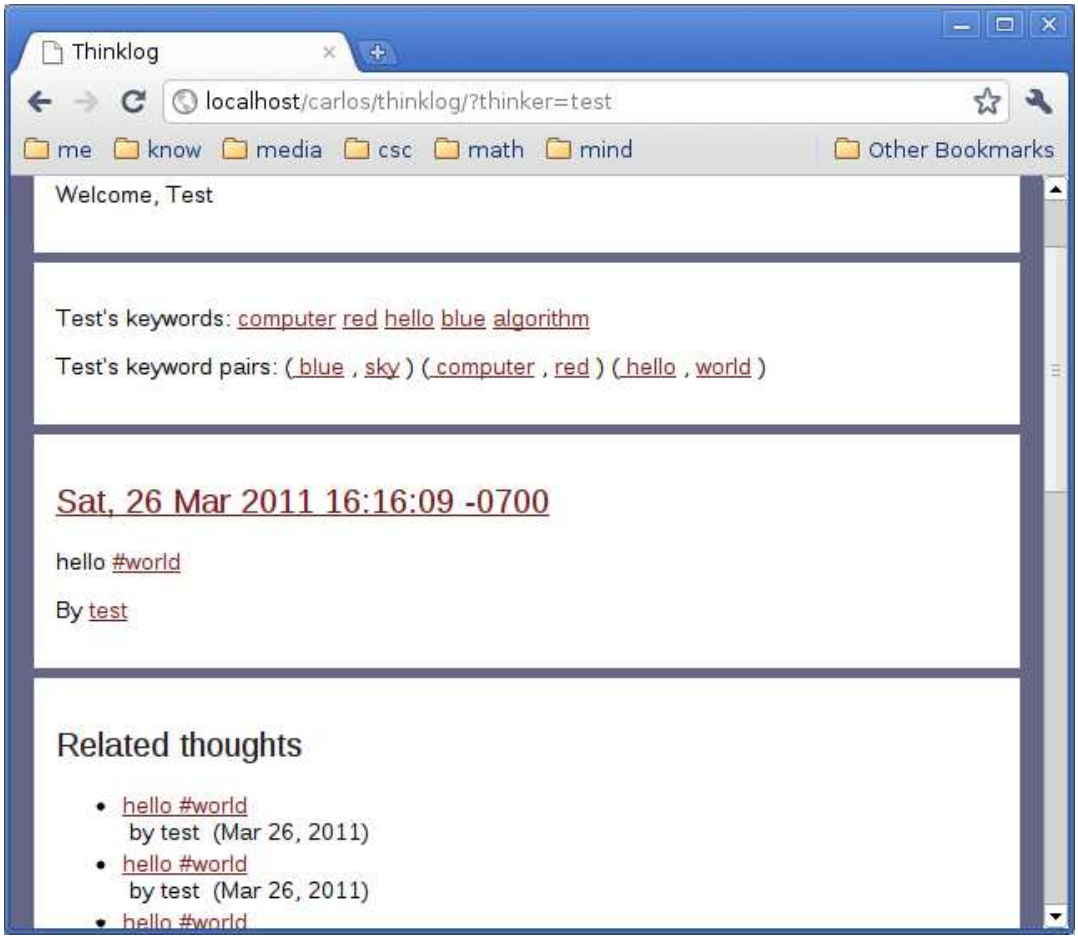
Since many users think out loud, so to speak, on Twitter, it is potentially a great fountain of thought, albeit short ones (only 135 or so characters per thought). Since Twitter delivers search results in RSS form, a user needs only to type Twitter's URL for the RSS feed into Thinklog's upload interface, and the application will get the RSS feed and upload the thoughts into the knowledgebase. However, not everybody knows how to get RSS feeds from Twitter. The following URL gets a query for tweets mentioning "computer science":

```
http://search.twitter.com/search.rss?q=computer+science
```

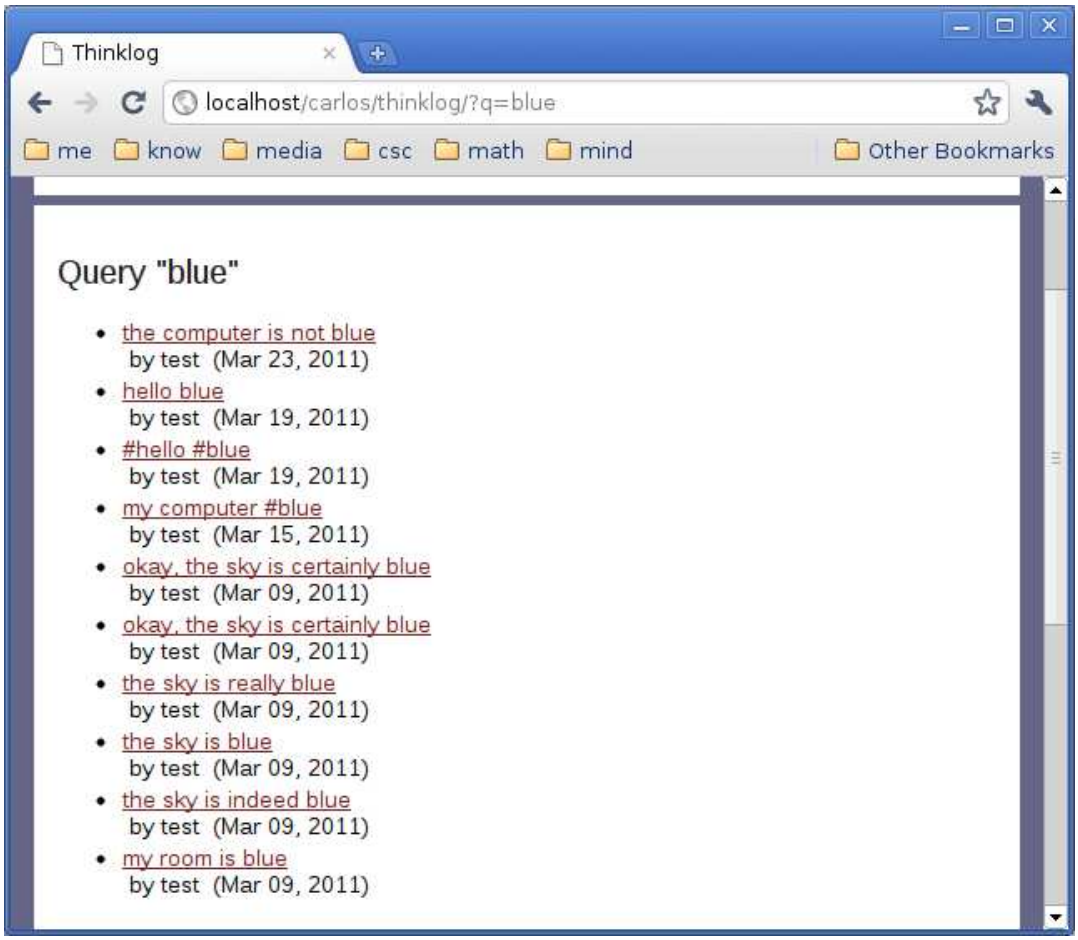
At the beginning of this project, it was hoped that we can have automatic streaming of thoughts from Twitter. The user would type a Twitter query, press a button, and the thoughts of that query would be downloaded from Twitter and stored and harvested in the knowledgedbase. Even though such functionality can be implemented by automatically fetching the RSS feed for a query, this would violate Twitter's terms of service, which says that Twitter content can't be output publicly by an application [5]. Instead, the user has to manually specify the RSS feed's URL, and it's the user's responsibility to not publish intellectual property that they do not own.

Screenshots of user interface

I give here a screenshot of the web application while viewing a thought and its tag cloud and related thoughts:



And below is a screenshot of a query for "blue". There is a glitch in the order of relevance of the search results, as "sky" is related to "blue", so thoughts with both "sky" and "blue" should come before other thoughts with just "blue".



Optimization

Since people have a lot of thoughts, we want to make sure the application can handle a lot of thoughts in the knowledgebase. The following discussion on performance of the Thinklog system is for a database of 50-200 thoughts. The server running Thinklog has a 1 GHz Intel Celeron processor with about 1 GB of memory.

Database and web server optimization

Currently, the application is coded in PHP, which is parsed and interpreted on the fly. SPARQL queries are parsed also on the fly, into SQL, and the user interface consists of many classes that are responsible for rendering HTML to display the application and interact with the user. As all this is happening each time a user requests a page, these factors could contribute to delays. But they are not the bottleneck. It seems that doing a query through the ARC 2 library is by far the most expensive operation, and in general decreasing the number of calls made to ARC 2 has made the application run faster.

However, other changes could be made to the server. Using a byte-compiled language like Java might be faster than using PHP. Also we could get precompiled SPARQL queries so that we don't have to translate them to SQL on the fly. Also Alex Thomo suggested that the calls ARC 2 RDF library may be making a connection to the database each time it does a query.

It is also possible to re-invent a "better wheel" and code an RDF triplestore library specialized for our operations.

Updating thought's mentions

Looking for which phrases are keywords of a thought is done both by the updater, and when the user adds the thought. If a thought has n words, then it has $O(n^2)$ phrases, because we take each contiguous sequence of words in the text. When looking for new mentions, we look at the texts of thoughts and see if there are hashtags or common keywords in the text. The triplification of each thought's mentions involves four SPARQL queries:

1. See if each phrase is a common keyword,
2. See if the keywords of the thought have nodes in the triplestore,
3. If not, then add the keyword nodes, and
4. Add the *mention* relationship for this thought and each keyword node.

It was found that caching a word when it is found to be a keyword, and caching whether or not a word is a common keyword, greatly reduces the time it takes to update the mentions for a thought, but only if it so happens that every word of a thought has been cached. This is because if some words have to be looked up, we still have one query to make. Since words are taken to be any contiguous sequence of words (or a phrase, like "am happy"), this is not likely to pay off. Thus, we check each thought's phrases to see if they are common keywords each time we process it during the update process.

Also, since a non-keyword can always become a keyword later, caching the fact that something is not a keyword defeats the purpose of finding this out. However, since keywords are often reused, it does help to cache whether or not a keyword is already in the knowledgebase so we don't waste time looking to see if it's there. A keyword will not suddenly become not a keyword (currently, no interface is set up to allow users to delete thoughts).

Updating keyword frequency

To find all keywords that are common, `updater` collects all the keywords in the knowledgebase and filters those that occur in at least the threshold-number of thoughts. Then it records the common keywords in the triplestore by inserting the triple (W, *rdf:type*, *commonKeyword*). Then, when we step through each thought and check for common keywords, we do not need to count the number of thoughts each keyword occurs in, because we just search for keywords that have the *CommonKeyword* type.

After finding which keywords are common, a routine can be run to find which words have pages in Wikipedia if they are not already common keywords in the system. As there are many words in all of the thoughts, and each query to Wikipedia takes about 3 seconds, this is too slow and is commented-out in the source code to reduce the total update process.

Updating keyword pair relationships

The `updater` also looks for relationships between keywords. This involves a query to find keyword pairs and the number of thoughts in which they occur together, and a query to add the keyword relationships for the pairs that occur in at least the keyword threshold number of thoughts.

For a database of about 200 thoughts, it takes about 94 seconds for the entire update process, from checking for common and related keywords, to updating the keywords each thought mentions.

Querying for relevant thoughts

When a user makes a query for thoughts relating to a set of words, only one query is made, a query for thoughts that mention these words or that mention words that are related to these query words. This is all done in one SPARQL query, and takes about 1.0 seconds for a two word query. To display a thought on the page, which includes a query for thoughts related to a particular thought, another query for retrieving thought content, and another query for the "About" section of the author of the thought, it takes about 3-4 seconds.

ARC 2 maintains B-tree indexes on the triple store for any subject, predicate, and object combinations, so it is expected that the keyword queries are efficient. However, the sorting of search results on the database side might require the entire database to be read. But querying for all thoughts, not relevant to any particular query, takes only 0.5 seconds. In comparison, searching for thoughts with words in common with the query requires also to group triples by each thought. Also search results are paginated in the SPARQL queries, so only 10 thoughts are fetched from the database per user request.

One idea to speed up retrieving thoughts related to a certain thought is to precompute the relevance between each pair of thoughts. This would take away the need for `GROUP BY` clauses in the SPARQL query for querying for thoughts that have keywords in common with a certain thought. We can reduce the number of triples we need to make for this by only recording the relationship if the number of common words is greater than zero.

Tag cloud: retrieving common keywords / pairs

To find the top keywords and keyword pairs for the 50 thoughts, the query to find keywords mentioned by the most thoughts and the related keyword pairs mentioned by the most thoughts takes together 1 second if we group by keywords and count the thoughts mentioning them. For seeing keywords related to a certain thought or a certain thinker, it takes about 2 seconds.

To shorten this time, I added routines to the updater that precompute the keyword and keyword-pair counts so that the tag cloud query does not need a `GROUP BY` clause. This shortened the time it takes by about one second. It still takes long because we have to sort the set of all keywords still. An idea to mend this would be to precompute what the most common keywords are for each thinker, which would make the update process longer.

With the precomputed counts, retrieving a tag cloud for the certain thought takes 0.4 seconds. Retrieving the tag cloud for a certain thinker takes 0.7 seconds. Retrieving the tag cloud for the entire set of thoughts takes 0.3 seconds. (The latter is so small because we don't have to join the count triples with the thoughts that mention them.)

Conclusions

Although I successfully created a system that found relationships between thoughts and keywords in an RDF triplestore, the system is quite slow. Perhaps using a more efficient interpreter like Java or optimizing the way the RDF library makes queries is the solution. On the other hand, given the length of time it takes to make queries that group by thoughts and sort by count, we might make a trade off that gives us less accurate estimate of thought relevance.

While keywords pertaining to our thoughts might help us become mindful of our tendencies, there may be other more useful information that can be inferred. I hope to improve the efficiency and usefulness of this application in the coming months, and maybe then I can implement it on the Internet for the world to use. Last time I checked, "`thinklog.net`" is available.

References

1. Mendes, Passant, Kapanipathi. Twarql, Tapping into the Wisdom of the Crowd. I-SEMANTICS 2010 September 1-3, 2010, Graz, Austria.

2. DBPedia. Data sets.

   http://wiki.dbpedia.org/Datasets (Accessed 2011-03-04)

3. Semsol. SPARQL+.

   https://github.com/semsol/arc2/wiki/SPARQL%2B (Accessed 2011-03-19)

4. Twitter.

   http://www.twitter.com/

5. Twitter. Terms of Service.

   http://www.twitter.com/tos (Accessed 2011-03-22)

Acknowledgements

Many thanks go to Alex Thomo for his guidance as my supervisor, Jon Muzio for the great seminar, and the audience of my presentation for their interest in my project.

Also many things drive me to provide this functionality to users. It is hoped that the ability to organize and relate thoughts will help people liberate themselves from lack of meaning in life. The key to life is thinking, so software that gets people to think has many applications.

Also Maya Rupert at TRU introduced me to folksonomy and the semantic web, giving me the initial spark to pursue this subject.

Finally, Dan Brown's *Digital Fortress* had a character (the crypto-chief) who had a program that organizes and relates his thoughts, which gave my first idea for this application.