Aug 2, 2006 Notes
Glenn Takanishi

Text Search
Strategies and Architectures

These notes view text searching strategies with respect to three specific application frameworks: Xapian[1], Redland[8] and 3Store[9].

The principle idea used in building the search tool, Xapian, was to keep the index as simple as possible. This is the main reason that keyword query resolution in Xapian is as fast as it. Pure speed enables the possibility of very large query resolution or multi-set keyword pattern matching.

There are always a multitude of technical trade-offs used to balance indexing speed and size with query resolution accuracy. Xapian's database is simple enough so that hundreds of thousands of documents can be linked together in a single index, and also simple enough so that dozens of separate indices can be processed simultaneously in parallel. The simplicity of the indices make possible the flexible configuration the Xapian architecture, and its optimal pattern matching performance.

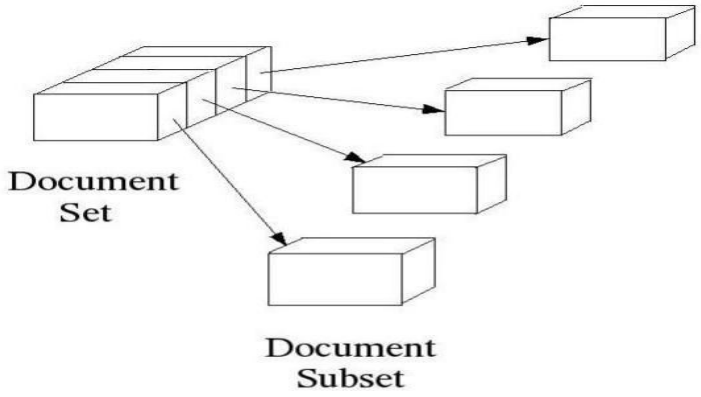
But multi-set pattern matching lacks software intelligence. Very complex, natural language style queries using inferencing or deductive logic invariably require building intelligent linkages to multiple documents in the dataset. In the past, most designers requiring software to execute complex queries using inferencing or logic put the intelligent linkages spanning the whole dataset into a single index. This makes parallel, distributed processing of the index practically impossible. On the surface this seems paradoxical, but in reality the design of text search software which superimposes logic or inferencing on top of plain pattern matching increases in complexity exponentially. This property of the exponential increase in complexity is called NP-completeness. This property is the beast we must tame.

Determining the context in which the document was written is an essential part of classifying the document. But the acquistion of context is intrinsically related to the rise of complexity. Context maybe acquired by semantic pattern recognition methods such as verb/predicate ordering, and entity/subject classifcation. As much as possible, we can inject context into the query resolution proceses by associating keywords with higher order action or predicate systems, and specific categories. The context for each document varies greatly. So, obviously, it is really hard to create software which can determine the context of each document by itself . But the RDF atoms or triplets contain the essential ideas for associating documents together using terms which link verbs or predicates with subject entities. At least, the RDF triples contain the essential elements for associating the basic semantic elements together.

1 Introduction

In a full text search tool, the design of the index affects everything: dataset load speed, index storage size,

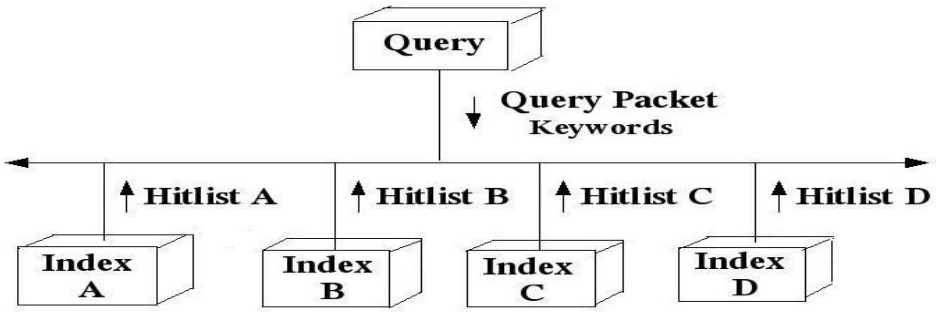
query response speed and query resolution accuracy. Instead of creating a single monolithic index like those used with advanced design[1] features such as a natural language interface, Xapian's indices simply map weighted keywords to documents. Xapian's simple design requires the domain of keyword pattern matching to be limited to each single document with no specially featured index linkages within each document. This constraint enables each document to be processed independently of each other. There is no chain of relationships between documents as they get indexed. The document set maybe divided naturally into separate index partitions which can be processed simultaneously in parallel. Each document is considered independent of each other.



Partition the dataset into smaller sets.
Index each subset separately.

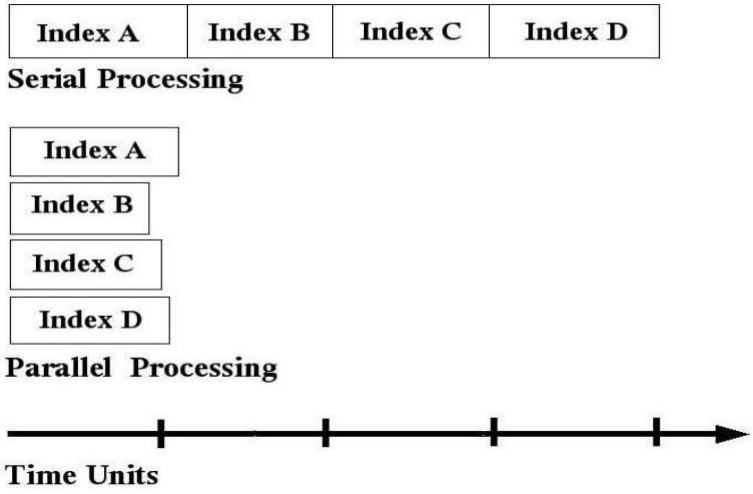
In a way, generic simple keyword search software is really simple because the designer is primarily focused on constructing the indices. Xapian features different index types such as Quartz and Flint[2]. One can index documents using both Quartz and Flint indices on separated document subsets, and then resolve queries using the hitlist returns from both types of indices.

1 The design of the index used in Infact [3], creates keyword linkages to potentially every sentence in every document in a single large index. Complex, multi-attributed keywords contain a hugh amount of linkages to each document object. As the size of the index data structure grows, the time required to add new text data increases exponentially. Since the complexity of the index is high, the index grows exponentially as documents are added to the database. Hence, it is almost impossible to design a way to update the index quickly.



Each processor on the network services one index. A query packet is sent to all processors. Each processor returns a hitlist it resolved, and the query processor adds up the returned the hitlist.

In parallel processing, the query resolution time is reduced by a factor of the number of index processors. If the number of index processors per query is 4, the query resolution time is one-fourth the amount of time required by regular serial processing.

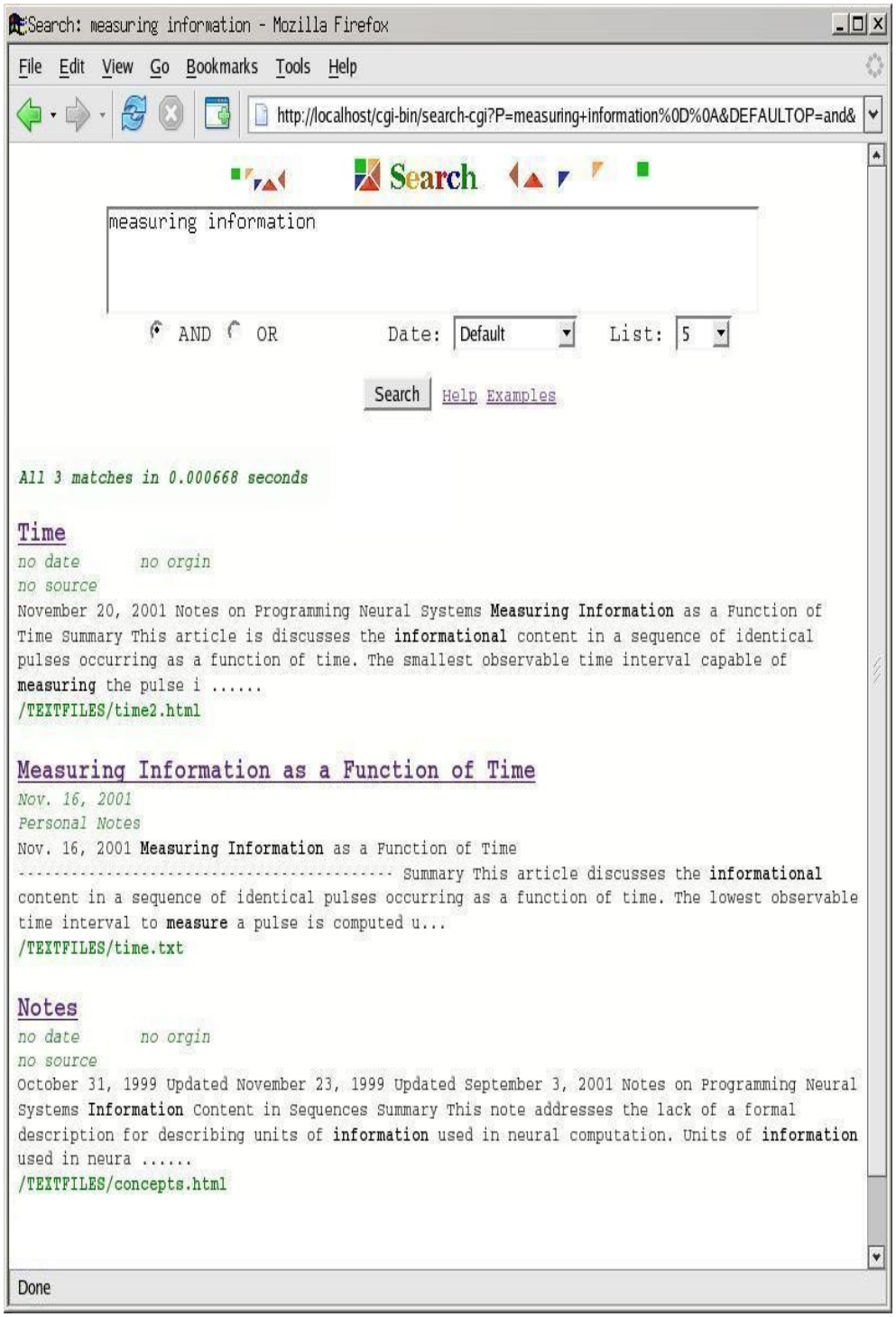


The general rule of thumb for keeping search speeds fast is to keep the btree indices relatively short. From experience with Xapian, for general text documents, an index size of 10 Gb is optimal. If the average size for a text document is about 2,000 characters, then the index mapping should be limited to under 500,000 documents. Generally, the uncompressed, working index maybe up to 20 times the size of the raw document set. For example, a typical Xapian index size containing 500,000 documents is about 10 Gb. The size of the index may vary greatly depending on the variety of keywords in the text document. Documents on specialized topics containing many unique terms will have enormous indices.

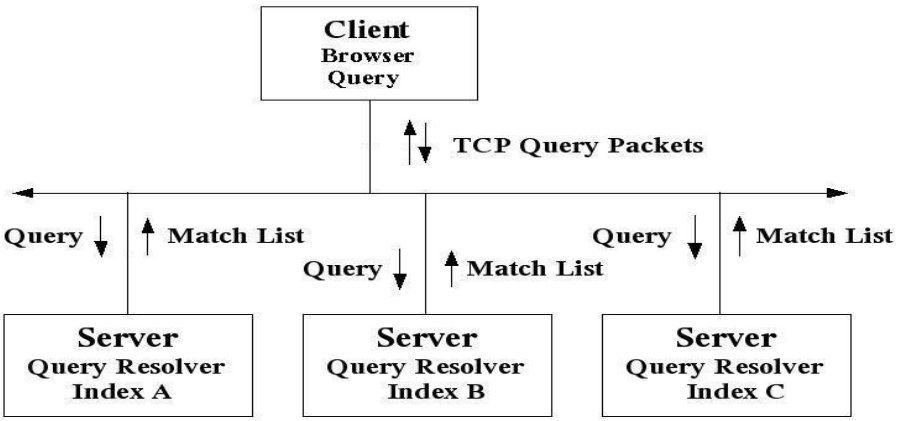
Computers with large memory spaces can be setup a with a RAM disk so that all the indexing can be performed in memory. Indexing and searching are I/O bound. The Sun multi-processor server, V880, has 32 Gb of RAM. If we kept the index size below 10 Gb, we could setup a couple of RAM disks on this server.

Xapian allows searching multiple indices simultaneously. The optimal configuration for a large document

set would be subdivided into dozens of subsets; each document subset using its own index. Omega is a Xapian client program which runs in a web browser as a CGI program. In a parallel, distributed network, the request for query resolution from the client Omega would be sent to mulitple resolvers which traverse separate indices: Index-A, Index-B, etc. The client sends the query resolution servers for Index-A, Index-B, etc., a TCP packet containing the same query string, eg., “measuring information” in the figure below. Each query resolver sends back to the client a TCP packet containing its hitlist for the query. The client Omega would integrate the returning hitlists from each resolver by relevance so that the final hitlist could be displayed in the brower.



Client program Omega running as a CGI program.



Client/Server Communication

Xapian's indices do not contain any linkages explicitly relating separate documents in the dataset together except semantically by common keywords. Theoretically, the query match list or hitlist does not contain any explicitly linked documents within the dataset. Separate documents are bound together semanically by the commonality of the keywords. This is an obviously important point, but sometimes overlooked. We can use this feature of semantically bounded documents in the hitlist as a filter to limit the size of the document set's domain of analysis. This significantly reduces the computational complexity due to the combinatorial explosion of keyword linkages in an index map for a large document set.

Combinatorial explosion in linkages occur in monolithic style indexes. The text search discovery tool, Infact[3], requires extremely large computing resouces to create its index. From experience, it would have been very difficult, maybe impossible with the computing resources we had used, to index a very large dataset containing over 10 million documents. Moreover, updating the index in time as the dataset changed could not have been done.

2 Search Algorithm

Document retrieval using keywords with Boolean logic is simple, and according to the documentation[2] on the Xapian site, this does not depend on the notion of relevance. "Boolean IR is so simple that it is almost possible to dispense with the notion of relevance altogether." However, the notion of relevance is central to the probabilistic model which is Xapian's principle retrieval model. "In the probabilistic model there is however a notion of relevance ..." The ranking of query returns depends on the significance or weight given to each of the keywords[4]. The practical implementation of the probabilistic model in Xapian was kept as simple as possible.

3 Adaptive Indexing

Keywords are weighted. The more significant keywords will determine the rankings of the query returns.

The semantic relationships between documents can be changed by reassigning different weights to keywords. Xapian was designed with a dynamic index which can be updated while user simultaneously queries the index. Documents which change over time can be resubmitted at any time to the Xapian index worker for update.

The relevance of query returns can be affected by the users through the queries they submit. Frequently used keywords in queries can be assigned higher weights, and the indices can be dynamically changed to reflect the interests of the users. In the Xapian software, this is called adaptive query scoring. The more popular results to be returned by the query are determined by the increasing the weight of frequently used keywords. In Xapian, the object called RSet, which stands for relevance set, lists documents as relevant, and modifies the keyword weights when performing queries.

Adaptive index structures[5] have further benefit of optimizing the index data structures such as its Btrees, and dynamically increasing seach performance over time.

4 Optimizing Query Resolution using Inferencing[2]

Query returns from searching a large document set maybe overwhelmingly large containing thousands of entries in the hitlist. The impact of large query returns has the effect of defeating the user's spirit because of the endless stream of documents requiring further filtering: either by manually reading each hitlist entry or creating more innovative queries to limit the number of returns. An ideal tool to help aid the the user to analyze the document set more efficiently is an inference engine. The RDF SparQL engine is based on first order predicate logic and graph theory. The RDF graph nodes or triples[6], are elementary atoms which can create linkages between text documents . As long as the number of linkages remain relatively small, on the order of under 5 million triples, the SparQL engine performs quickly enough. A dataset of less than 5 million items would be a natural fit for an SQL database[7] running on a single processor .

Most RDF databases such as Jena or Sesame use open source databases such as Postgres or MySQL to store the RDF triples. The 3Store[8,9] is a simple and efficient RDF database which could worked nicely with Xapian.

The user would have to populate the triple store database by submitting a set of queries first before using the inference engine. This is much like training the inference engine on what type of document set the user is interested in.

5 Summary

In theory, the text search engines like Xapian, Lucene, and Yahoo and Google are similar. They operate on the same software foundations. The Yahoo and Google search engines are highly distributed using

2 Enabling inferencing services is the key to better query resolution[10]. The following is a quote from Stephan Decker's article. “The creation of RDF raises the prospect of a widely accepted standard for representing expressive declarative knowledge on the Web. However, just representing knowledge and information is not enough: users as well as information agents have to query and use the data in several ways. An RDF Query specification would allow a range of applications to exploit any information source which can be represented in terms of the RDF data model.”

thousands of tightly coupled computers on a network to achieved their blazing speeds. The next step up on the software scale is adding intelligent inferencing to these search engines.

It seems that to a degree, the RDF triple stores which are able to perform predicate logic operations on its RDF graphs are the natural next step up the scale in the search engine's evolution. But the level of detail in implementating inferential and transitive entailment capability on top of a fast textual database is very large. Much of the triple store software such as the SparQL query engine works. However, the implementation of the SparQL query engine has taken a painstakingly long while.

We haven't covered enough examples showing query retrieval accuracy. But this is largely tied to integrating the 3Store SparQL engine with Xapian. The 3Store software code which executes the SparQL query resolver is only 2 pages long. However, I haven't been able yet to find a natural way to interface the SparQL engine with the Xapian query resolver.

Also our inference engine would need a concept extrapolator or propagator. The Open Mind Common Sense software called OMCSnet helps the inference engine to follow possible legitimate concepts related to the searched text documents.

I'll try to write up more about this next month in September.

References

1. Xapian Home Page, http://www.xapian.org/
2. Xapian documentation, http://www.xapian.org/docs/intro_ir.html
3. Infact, http://www.insightful.com/products/infact/default.asp
4. Information Retrieval, by C. J. van Rijsbergen, http://www.dcs.gla.ac.uk/Keith/Preface.html
5. Adaptive Index Structures, Yufei Tao and Dimitris Papadias, Proceedings of the 28th VLDB Conference
6. RDF Triples, http://www.w3.org/TR/rdf-concepts
7. MySQL Home Page, http://www.mysql.com
8. Redland RDF Store, http://librdf.org/
9. 3Store Web Page, http://www.aktors.org/technologies/3store

10. A Query and Inference Service for RDF; Stefan Decker , Dan Brickley, Janne Saarela and Jürgen Angele;
http://www.w3.org/TandS/QL/QL98/pp/queryservice.html

11. John McCarthy, Notes on Formalizing Context:
http://www-formal.stanford.edu/jmc/context3/context3.html

12. M. Ross Quinlan, Semantic Memory: 1968, Semantic Information Processing, M. Minsky (ed), 216-260, MIT Press

13. RDF Vocabulary Description Language 1.0 - RDF Schema: http://www.w3.org/TR/rdf-schema/

14. Stephan Kokkelink: Quick Introduction to RDFPath:
http://zoe.mathematik.uni-osnabrueck.de/QAT/RDFPath/Quick.pdf

15. Andreas Harth, Stefan Decker;
Optimized Index Structures for Querying RDF from the Web:
http://sw.deri.org/2005/02/dexa/yars.pdf

16. Dave Beckett, Redland RDF Application Framework – Contexts:
http://librdf.org/notes/contexts.html

17. A Universal RDF Store Interface: http://rdflib.net/store/

18. Philip McCarthy, Search RDF data with SPARQL:
http://www-128.ibm.com/developerworks/library/j-sparql/#1

Appendix A

1 Inferencing

One of the main problems of finding patterns in textual information is isolating the context[11] in which each sentence is expressed. Isolating context is a constantly occurring problem when reading, querying and making inferences in textual data. A semantic network[12] is constructed so that each word in the network gets its meaning from its association with other words in its neigborhood proximity.

In the semantic network model, the relative positions of triples[13] with respect to each other in the network defines what each triple is. The context in which a triple is defined depends on the other triples in its local neigborhood. Context can further be determined by interpreting reference markers for each triple[3] when traversing paths the graph network.

Stefan Kokkelink[14] says, "The purpose of RDFPath is to localize information in an RDF graph. It

3 Triple RDF reification or actuating the functor for the triplet descriptor class.

provides for a general technique to specify paths between two arbitrary nodes of an RDF graph." This notion that the contexts will reveal itself as one dynamically traverses network structures is important for making inferences. This essential feature of a query language is not explicitly implemented in SPARQL using explicit schemas, or recursive or nested forward and backtracking search algorithms. Instead, since SPARQL performs pattern matching on graphs or subgraphs, we essentially have to do "path navigation" by refining the pattern matching process submitting better input graphs by "trial-and-error" so to speak. This strategy keeps the algorthmic processes simple.

Brian McBride says,[4] "I earlier suggested that SPARQL does not have to support transitive closure because the graph can do it. ... The question arises, whether there is a need to distinguish between the direct relationship and the closure relationship. ... SPARQL supports querying over multiple graphs and so can support querying the ground graph to get at the direct relationship and an inferred graph to get at the closed one."

There are pros and cons to this: If you know what you want to query for, okay. However, if the you're seeking to discover patterns you cannot be expected to foresee, then maybe using a heuristically guided search strategy is better. These issues are related to using the semantic network, snet, software which helps in generating "context paths." Snet will have the effect of limiting the scope of the selection paths.

2 Context Nodes

A triple (s, p, o) is a directed link between two nodes in a network. A set of triples makes a graph. In the mechanics of navigating or traversing graphs, it's important to understand how much information which determines the traversal paths are missing[5]. The meaning of each isolated triple maybe ambigous because it can have multiple meanings. So traversible graph paths depends not only on the relative positions of the triples in the graph network, but on something more called **context**[6].

In the RDF data model's indexing [15,16,17] architecture, context is defined within a pair (t, c) where t = (s, p, o) triple, and c is a node (URI reference or literal) in RDF space. The context in RDF space may also be defined as a named subgraph[7] which is itself composed of a set of triples. The context is a coordinate in the graph.

---

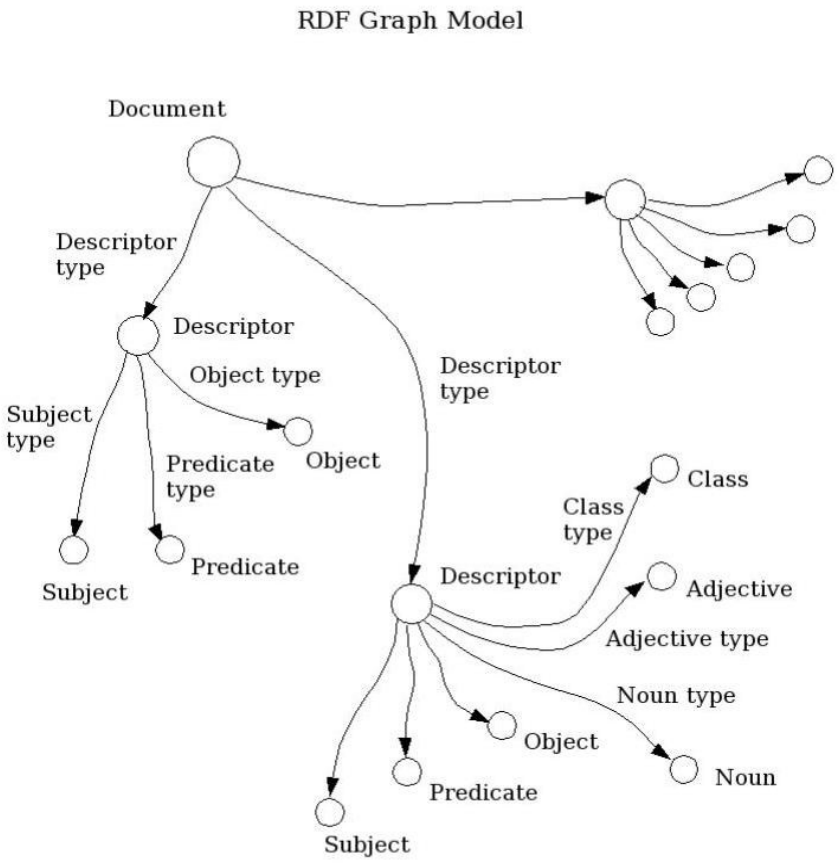
4 Brian McBride's email, Closure and SPARQL: http://lists.w3.org/Archives/Public/public-swbp-wg/2005Dec/0003.html

5 In interpersonnal, verbal communication, we usually assume the context of our information exchange without explicitly stating it. This overflows into the text domain.

6 Theoretically, context parameters would not be needed if there were a enough triples in the semantic network to describe or give meaning to each adjacent triple. That is, one could add context information to the RDF graph network by adding more triples. With enough triples you would have enough context to make decisions traversing graph networks.

7 Bizer, Carrorl; Modelling Context using Named Graphs:

http://lists.w3.org/Archives/Public/www-archive/2004Feb/att-0072/swig-bizer-carroll.pdf



2.1 Example Query

SPARQL can query multiple named graphs in one query. The following query[18] finds people described in two named FOAF graphs.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?name
FROM NAMED <phil-foaf.rdf>
FROM NAMED <jim-foaf.rdf>
WHERE {
  GRAPH <phil-foaf.rdf> {
    ?x rdf:type foaf:Person .
    ?x foaf:name ?name .
  } .
  GRAPH <jim-foaf.rdf> {
    ?y rdf:type foaf:Person .
    ?y foaf:name ?name .
  } .
}
```

I could not get the exact form of the query above to work using Redland's Rasqal query tool. However, a simpler form [***figure 1***] worked. The RDF graphs phil-foaf.rdf and jim-foaf.rdf were downloaded from

different web sites, but both contained the name and mbox-sha1sum field types.

A query can use named graphs as pattern matching templates against the search data. The following query[8] from Philip McCarthy's article show a named query used as a template on RSS data.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rss:  <http://purl.org/rss/1.0/>
PREFIX dc:   <http://purl.org/dc/elements/1.1/>

SELECT ?title ?known_name ?link
FROM <http://planetrdf.com/index.rdf>
FROM NAMED <phil-foaf.rdf>
WHERE {
        GRAPH <phil-foaf.rdf> {
          ?me foaf:name "Phil McCarthy" .
          ?me foaf:knows ?known_person .
          ?known_person foaf:name ?known_name .
        } .

        ?item dc:creator ?known_name .
        ?item rss:title ?title .
        ?item rss:link ?link .
        ?item dc:date ?date.
      }
ORDER BY DESC[?date] LIMIT 10
```

3 Comments

Using named graphs as pattern matching templates to extract data with specific contexts is a straight forward methodology. In some real world applications, this methodology reduces the complexity of pattern matching by using brute force recursive algorithms.

But if we start attacking pattern recognition problems which uses algorithms to discover new patterns, the RDF querys can become monsterously complex.

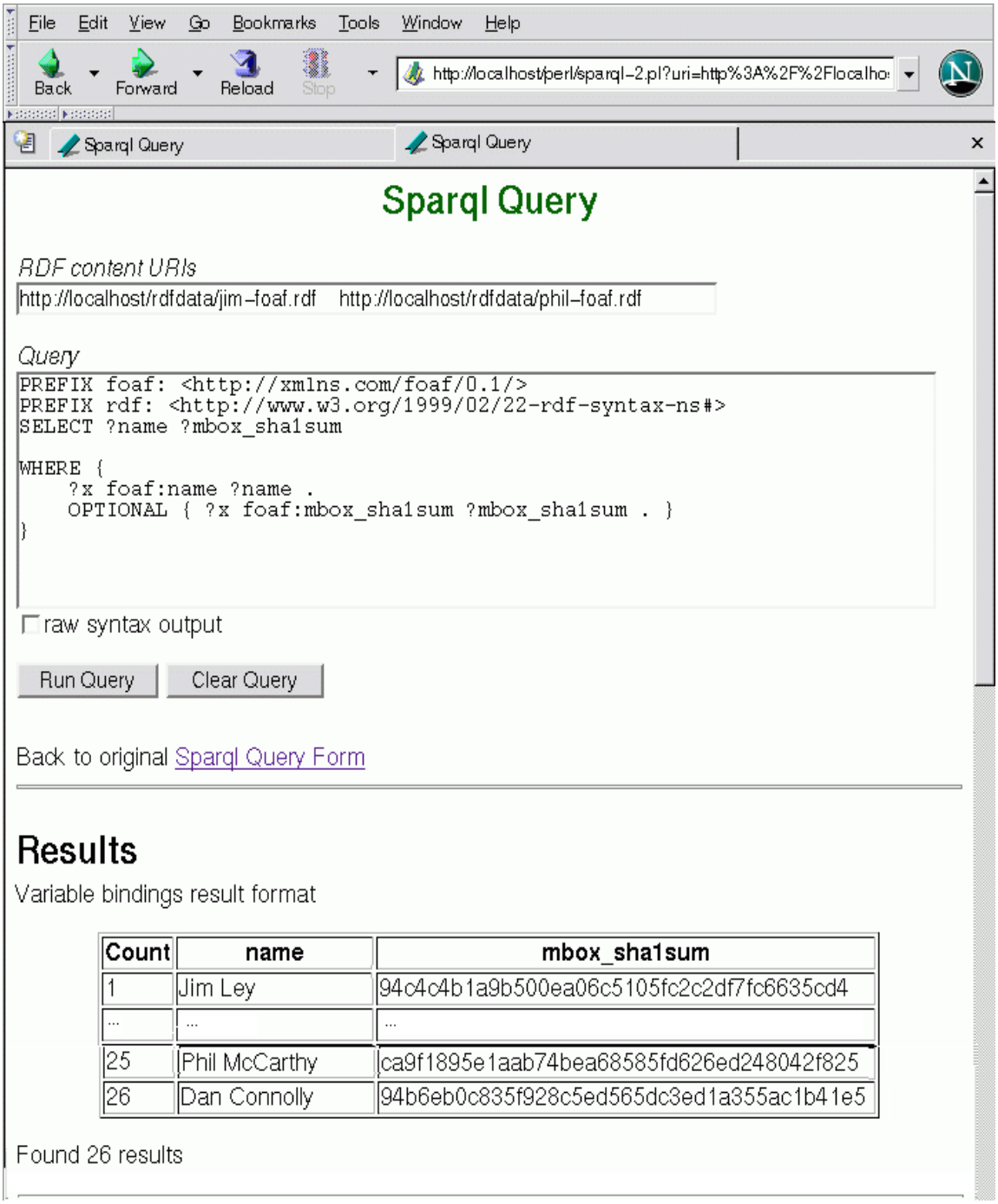


Figure 1. SPARQL Query on 2 Graphs