Semiometrics: Applying Ontologies across Large-Scale Digital Libraries

Duncan M. McRae-Spencer[1], Nigel R. Shadbolt[1]

[1] School of Electronics and Computer Science, Highfield, Southampton, SO17 1BJ, UK
{dmms03r, nrs}@ecs.soton.ac.uk

Abstract. As large-scale digital libraries become more available and complete, not to mention more numerous, it is clear there is a need for services that can draw together and perform inference calculations on the metadata produced. However, the traditional Relational Database Management System (RDBMS) model, while efficiently constructed and optimised for many business structures, does not necessarily cope well with issues of concurrent data updates and retrieval at the scale of hundreds of thousands of papers. At the same time the growth of RDF and the increasing interest in Semantic Web technologies perhaps begins to present a viable alternative at a scalable, practical level. This paper considers a specific application of large-scale metadata analysis and conducts scalability tests using real-world data. It concludes that RDF technologies are both a scalable and performance-realistic alternative to traditional RDBMS approaches. It also shows that for relationship-based queries on large-scale metadata stores, RDF technologies can significantly out-perform traditional RDBMS approaches by allowing both retrieval and updating of data in a timely manner.

1 Introduction

The emergence of large-scale online digital libraries is a feature largely welcomed by the academic scientific community. While systems such as Citeseer[1] and Google Scholar[2] crawl the web searching for papers, increasingly online institutional repositories are being created, exposing their papers and metadata in a standard format. These systems are sufficiently successful to have raised the expectations of the user community: it is now the case that people expect academic papers to be findable and downloadable, fully indexed and searchable in ‘Google’ style; citations to other documents should be rendered as hyperlinks; metadata should be searchable and services summarising the work of an author, institution or journal/conference should be made available. While the various digital libraries attempt to meet some or all of these expectations, it remains the case that the number of papers indexed and stored by these libraries is in the order of hundreds of thousands and will only increase as the move towards more open archiving[3] continues and more metadata becomes available. Producing services that run over these libraries, and perhaps even across multiple libraries, is therefore a challenge when considering the issues of search speed and query complexity.

At the same time, the growth of Semantic Web technologies may provide an answer to at least some of the questions raised above. The push towards more intelligent, computer-readable websites has brought to the fore the use of ontologies as a means of data manipulation and integration, and RDF as a format for data storage and transfer. While much semantic web research focuses on the development of storage techniques such as 3Store[4] and Jena[5] as well as inference-based language standards such as OWL, it is clear that RDF-based triplestores, along with the query language SPARQL, allow a different approach to be taken to data storage and searching than that which is provided in more traditional RDBMS models. This paper details the theory and practice of applying the RDF technique to large-scale digital libraries and shows how, for many more complex queries demanded by the raised expectations of services described above, data storage in RDF and querying by the standard RDF query language SPARQL provides a level of performance at least as useful as standard SQL approaches, and fast and flexible enough to provide a real option for use in online digital library services.

2 Motivation

The relational database model, queried by SQL, has been a standard model for data storage for many years. While optimisation and indexing techniques have boosted the efficiency of this model, it remains the case that some queries on multi-table databases remain complex even though they are easily expressible in plain language. For example, given a simple database schema for a large metadata repository, some valid queries might be: ‘how many distinct authors are there in this system’, ‘which papers cite papers by this author’ and ‘what are the titles of the articles has this author written since 2002’. In SQL, these could be respectively expressed as:

```
1. SELECT COUNT(*) FROM authors;

2. SELECT DISTINCT bibliographies.MasterArticle,
 bibliographies.ArticleCited
FROM bibliographies INNER JOIN author
ON bibliographies.ArticleCited = author.documentID
WHERE author.AuthorID = 'P123';

3. SELECT DISTINCT articles.Title, articles.articleID
FROM articles INNER JOIN authors
ON articles.articleID = authors.ArticleAuthored
WHERE authors.acmID = 'P123'
AND articles.Year > 2002;
```

While the first of these queries is relatively simple, the second and third both involve inner joins, the third on a potentially very large table ‘articles’, raising query complexity and potentially increasing the time taken to produce a result, depending on the indexing techniques used. By contrast, these two queries can be expressed relatively simply in SPARQL, given a suitable ontology (in our case, we used a standard academic papers-and-people ontology created by AKT[6]).

```
2. SELECT distinct ?p ?c
WHERE
  {
   ?p akt:has-author <http://citeseer.ecs.soton.ac.uk/#P123> .
   ?c akt:cites-publication-reference ?p .
  }

3. SELECT distinct ?p ?t
WHERE
  {
   ?p akt:has-author <http://citeseer.ecs.soton.ac.uk/#P123> .
   ?p akt:has-title ?t .
   ?p akt:has-date ?d .
   ?d support:year-of ?y .
   FILTER (?y > 2002)
  }
```

While these queries may appear similar in terms of number of lines, the actual logic involved is far simpler in the SPARQL queries, and as will be shown in this paper, response times are greatly reduced. However, it is wrong to suggest that SPARQL is simply better than SQL: the first query is actually far better in SQL than in SPARQL:

```
1. SELECT distinct ?a
WHERE
  {
   ?p akt:has-author ?a .
  }
```

Despite the relative simplicity of the statement, there are two major problems with this query. Firstly, the query itself doesn't actually answer the question of 'how many' – SPARQL does not contain an equivalent of SQL's count(*) operation, and so the user (or the program making the call) would have to do the summation calculation separately. Secondly, and more importantly, this SPARQL statement has to query the entire Knowledge Base, finding all instances of the 'has-author' predicate, then creating a distinct list of the subjects of those triples. This is an extremely inefficient way to simply count all the instances of authors – however the nature of RDF means that we need to count the instances of the relationship in order to discover the identity of the URIs concerned: they are defined as being authors because they are subjects of triples whose predicate is 'has-author'.

3 Data Storage Models and Purpose

The essential difference between the RDBMS and ontology-based data models are their respective purposes. This section discusses the design rationales behind the two approaches and where the essential differences lie.

Relational databases typically deal with questions of identity, including if that identity involves calculations across tables. RDBMSs are optimised to allow efficient querying of data, data which is itemised in tables and columns according to identity.

This means that in practice, queries such as retrieving the total number of authors is straightforward – it is simply a summation of the number of distinct rows in the 'authors' table. However, queries based around relationships between data are more complex – although the relational model makes these queries possible, for large-scale databases with complex tables containing several hundred thousand rows it can be very time-consuming to perform the required JOIN operations.

To overcome this problem, RDBMSs typically offer users the opportunity to perform indexing operations on their data. User-chosen indices allow storing of sorted columns (or column combinations) meaning a vast reduction in search time, particularly when performing the more complex relational operations. The down-side of this is an increase in the time taken to perform inserts and updates to the system, as the indices associated will have to be updated. Additionally, for large multiple-indexed tables, the index files often grow to the extent that they become bigger than the actual database files they are indexing. For most systems, a trade-off can be made between the amount of indexing and the need to keep the system 'open' so additions and changes can be made as well as efficient querying: however, as described below, as systems become larger, the trade-offs become harder to make.

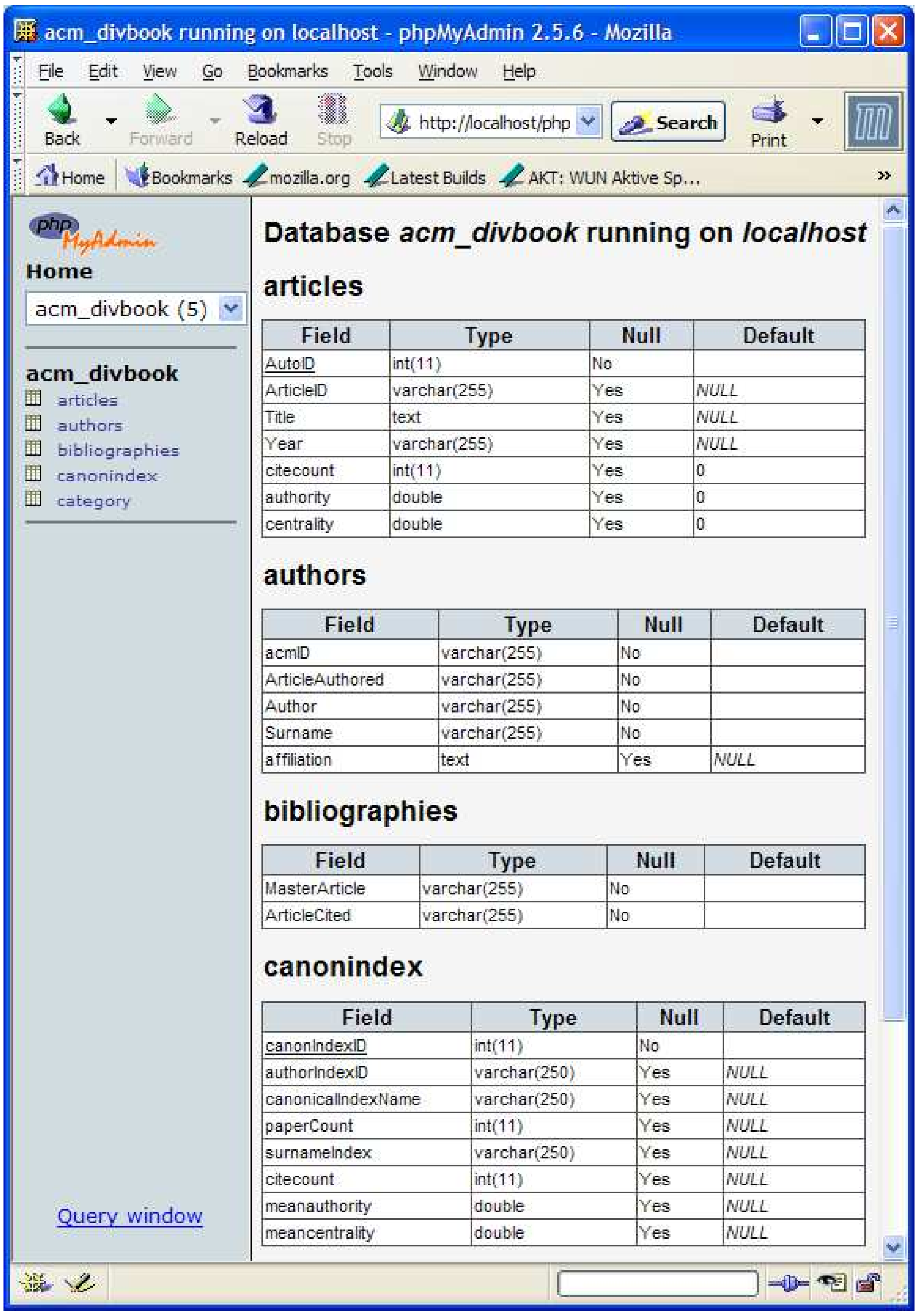
In contrast to the 'identity' model of traditional RDBMS databases, ontology-based data is designed to deal primarily with questions of relationships, where the predicates are the focus of the query. The emergence of RDF as a standard format for data description, coupled with the development of scalable triplestore solutions (such as 3Store in our case), has allowed the creation of searchable knowledge bases where relationship-based queries can be easily framed, provided the ontology concerned is sufficiently engineered to allow for such queries. In practice, therefore, queries such as retrieving the titles of all documents a particular author has written since 2002 is straightforward the system just needs to look for all the predicate-subject combinations where the has-author predicate is followed by the particular URI representing the given author, then filter out all results from 2002 and before. As we are essentially searching for a relationship rather than a set of answers from a table, the ontology model is suited to allow us to search for such information.

As a side-note, it is important to remember that underneath triplestores is usually a database of some description – indeed 3Store is built on top of the relational database MySQL, optimised with its own indexing. The various experiments described in the following section compare the relative efficiencies of the SQL and SPARQL approaches – but it is important to note that the SQL database used by 3Store and the one used in the experiments was the same MySQL installation on the same computer: the tests therefore were focusing not on the relative performances of databases, but on the differences between the two data models.

4 Practical Usage: Semantic Web Services and Semiometrics

The initial motivation for storing large-scale document repository metadata in RDF format came from the desire to produce usable, efficiently searchable services based on metadata from two computer science centric repositories: Citeseer and the ACM Digital Library. While straightforward searching and browsing facilities are fully implemented on the respective websites of these repositories, the desire was to provide more in-depth services based on data relationships, such as ‘influence’ scores for papers, authors and institutions based on more than purely citation counting alone. To this end, the raw metadata (essentially Dublin Core[7] plus citations) was taken from the two sources and put into two different databases with identical schemas, as shown in figure 1. This schema, while containing a number of tables, was optimised to give the simplest possible view of the data in the smallest number of tables possible, while adhering to the basic relational database model. Thus there are three main tables: articles, authors and bibliographies, with a fourth (canonindex) introduced to help speed up certain author-based queries, even though this means a duplication of author data. Note: throughout these experiments there was no attempt to merge the two datasets as it was considered most useful to see how similar results would be across two completely distinct, although similarly sized, datasets.

Initially questions of indexing were answered by attempting to find a sensible trade-off between the need for indexing and the need for flexibility in terms of amending and, particularly, adding data. However, it quickly became apparent that while indexing allowed for quick searches, the indexed column and table became difficult to update with new and amended data in a live environment, even if such updates were stored up and scheduled for a low-usage period. The more indices, the slower the updates, even if the tables were otherwise optimised and non-essential features (such as foreign key constraints and cascade functions) were removed from the database and handled at the application level. On a small subset of Citeseer data, containing roughly 12,000 papers, a compromise model was possible containing a degree of indexing while still allowing for changes to be made to the database. For the full datasets, however, containing metadata, author and bibliography information for over half a million papers, no such compromise was possible: either the unique columns in the tables were indexed, effectively preventing regular updating, or they were not indexed, dramatically slowing search time. Eventually two models were chosen for the system: a ‘closed’ system with heavily-indexed tables that would not be updateable in a ‘live’ setting and an ‘open’ system with minimal indexing where updates could be made at the expense of search time.


acm_divbook running on localhost - phpMyAdmin 2.5.6 - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop http://localhost/php Search Print
Home Bookmarks mozilla.org Latest Builds AKT: WUN Aktive Sp...
Home
acm_divbook (5)
acm_divbook
articles
authors
bibliographies
canonindex
category
Query window

Database *acm_divbook* running on *localhost*

articles

Field	Type	Null	Default
AutoID	int(11)	No	
ArticleID	varchar(255)	Yes	*NULL*
Title	text	Yes	*NULL*
Year	varchar(255)	Yes	*NULL*
citecount	int(11)	Yes	0
authority	double	Yes	0
centrality	double	Yes	0

authors

Field	Type	Null	Default
acmID	varchar(255)	No	
ArticleAuthored	varchar(255)	No	
Author	varchar(255)	No	
Surname	varchar(255)	No	
affiliation	text	Yes	*NULL*

bibliographies

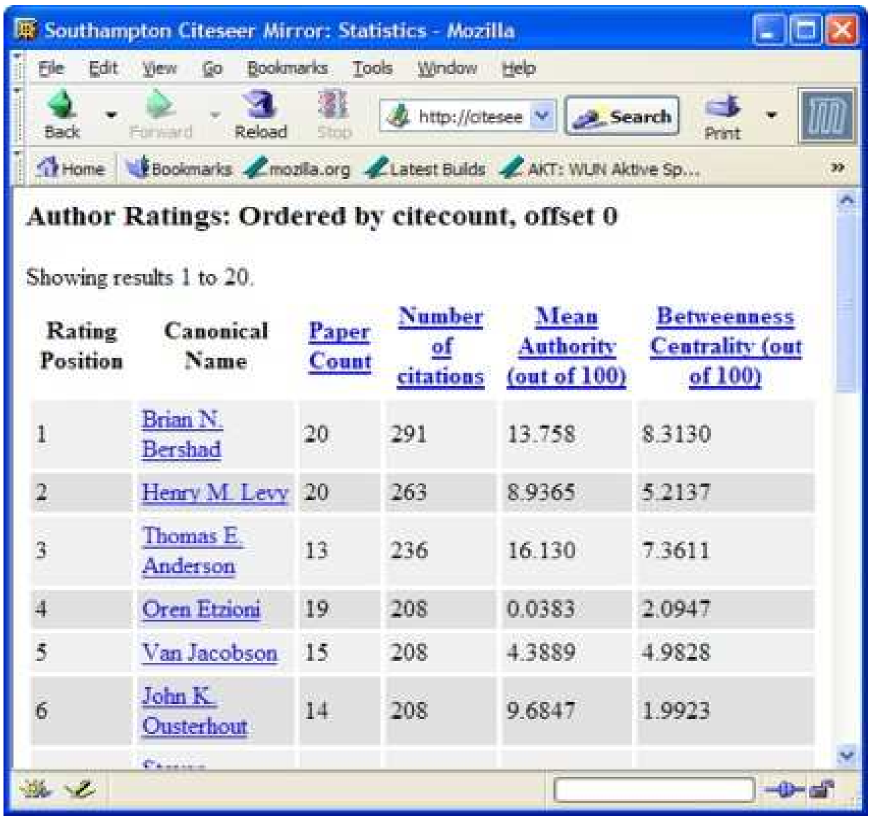
Field	Type	Null	Default
MasterArticle	varchar(255)	No	
ArticleCited	varchar(255)	No	

canonindex

Field	Type	Null	Default
canonIndexID	int(11)	No	
authorIndexID	varchar(250)	Yes	*NULL*
canonicalIndexName	varchar(250)	Yes	*NULL*
paperCount	int(11)	Yes	*NULL*
surnameIndex	varchar(250)	Yes	*NULL*
citecount	int(11)	Yes	*NULL*
meanauthority	double	Yes	*NULL*
meancentrality	double	Yes	*NULL*

Fig. 1. The database schema for the MySQL sources for the Citeseer and ACM datasets. Note that ‘canonindex’ is a short-cut index table for storing pre-calculated information on authors to speed up query times.

Using these models the metadata, along with a few of the more in-depth results, were exposed through a number of web services and utilised by two separate client systems. The first of these was a set of web pages made available via a local server alongside a mirror of the existing Citeseer system, provided by Penn State University. These pages utilised many of the web services to provide a coherent set of results users would be able to search and browse. Initially implemented using the Citeseer data subset of 12,000 papers held in the 'compromise' index model described above and shown in fig. 2, the system was expanded to the full dataset using the 'closed' model described above after the 'open' system led to more time-outs than actual results being displayed. While the 'closed' system was sufficiently quick to respond to queries, and thus useful for demonstration purposes, it was clear that in practice a system that was effectively 'frozen' would not be useful in anything other than the very short term. For the remainder of this paper, we will use the terms open and closed SQL databases to refer to the databases produced with minimal and heavy indexing respectively.


Southampton Citeseer Mirror: Statistics - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://citesee
Search
Print
Home Bookmarks mozilla.org Latest Builds AKT: WUN Aktive Sp...

Author Ratings: Ordered by citecount, offset 0

Showing results 1 to 20.

Rating Position	Canonical Name	Paper Count	Number of citations	Mean Authority (out of 100)	Betweenness Centrality (out of 100)
1	Brian N. Bershad	20	291	13.758	8.3130
2	Henry M. Levy	20	263	8.9365	5.2137
3	Thomas E. Anderson	13	236	16.130	7.3611
4	Oren Etzioni	19	208	0.0383	2.0947
5	Van Jacobson	15	208	4.3889	4.9828
6	John K. Ousterhout	14	208	9.6847	1.9923

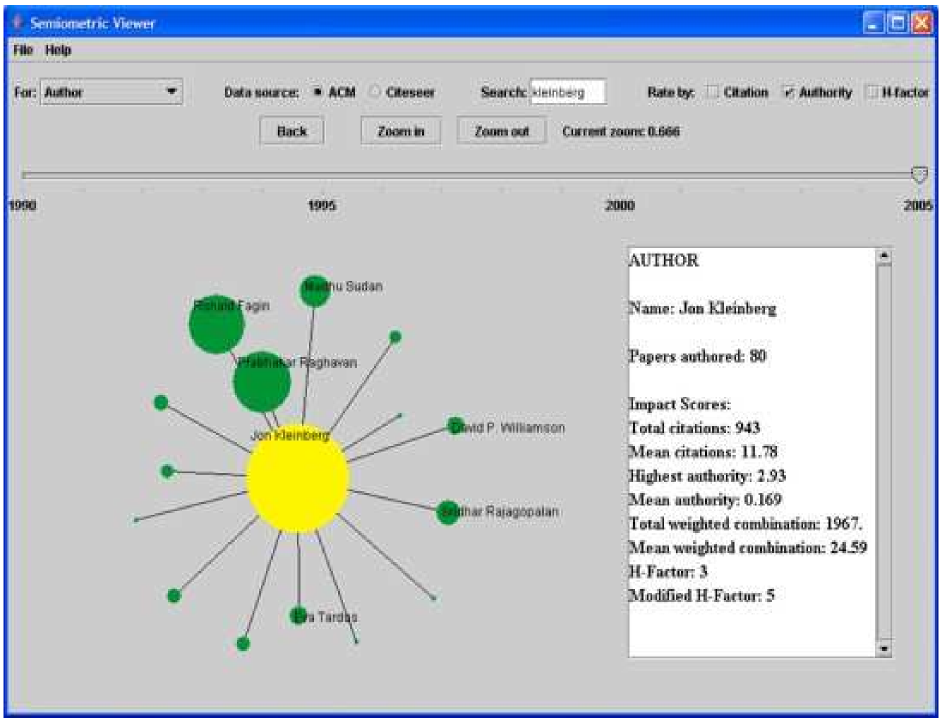
Fig. 2. Screenshot showing the semantic web services client pages running on a sub-set of Citeseer data.

At this point the direction of the work was switched to see if the RDF/ontology model would provide any answers. Although theoretically, as described above, it was clear that a SPARQL-based set of queries to a triplestore would provide a different set of response times for the same results, it was unknown whether the increase in efficiency over the ‘open’ SQL model would prove sufficient to be able to offer the services we wanted in a reasonable timescale. Similarly it was unclear whether SPARQL alone would be able to provide all the answers, given the examples in section 2 of this paper which showed the clear advantage of SQL in identity-based queries: would a combination of SPARQL and SQL be better?

The first client, the set of web pages parallel to a local Citeseer mirror, were therefore re-written to utilise (also re-written) SPARQL-based web services. With paper metadata populating the AKT Reference Ontology[7] and with the Citeseer data augmented by our automated disambiguation tool AKTiveAuthor [8], the services were able to query the data asserted in a 3Store v3 running a number of other KBs, and on a server running numerous other web applications including the Citeseer mirror.

The results were encouraging: for the majority of searches translated into SPARQL, the searches completed in a suitable time for use in web services. The few that were too slow matched the few that SQL queries had proved capable of responding to in a reasonable timeframe from the open database. Therefore overall success of the SPARQL semantic web service querying model, combined with a small number of SQL queries to an open database, allowed the development of the next stage of the process: the creation of a Semiometric viewer application.

The SemioViewer application uses, as described above, a combination of SPARQL and SQL queries, calculating influence scores for papers and authors on the fly, producing summary data for selected paper/author, search interface and browsing of neighbouring papers/authors (citations for papers and co-authors for authors), whose influence scores are also shown. The prototype application (shown in fig. 3) is written in Java and calls a variety of SPARQL queries via HTTP, as well as the equivalent SQL queries directly to MySQL (querying either the open or closed database). The overall purpose of the application, in conjunction with the web services described above, is to allow the browsing and calculating of influence scores at various levels of granularity – papers, authors, institutions, disciplines and others. While the theory and results of that work is discussed elsewhere[9], the SemioViewer provides a platform for comparing the SQL and SPARQL approaches. As it contains equivalent queries in both languages, and as the application is designed to be used as a practical interface to large-scale metadata stores, it proved to be an ideal ‘test ground’ to compare equivalent SQL and SPARQL queries into data stores dealing with several hundred thousand papers.



For: Author

Data source: [✓ ACM] Citeseer

Search: kleinberg

Rate by: Citation [✓ Authority] H-factor

Current zoom: 0.666

AUTHOR

Name: Jon Kleinberg

Papers authored: 80

Impact Scores:
Total citations: 943
Mean citations: 11.78
Highest authority: 2.93
Mean authority: 0.169
Total weighted combination: 1967.
Mean weighted combination: 24.59
H-Factor: 3
Modified H-Factor: 5

Fig. 3. Screenshot of SemioViewer application showing search on ACM dataset, revealing impact scores for Jon Kleinberg and the relative closeness and impact of his co-authorship community.

5 Results

The purpose of this paper (and of these experiments) is not to see how good Semiometrics are: instead the purpose here is to ask two key questions: (1) is it useful to replace open SQL queries with SPARQL for some/all queries, particularly where SQL is very slow and (2) is it a realistic hope to produce a system that works in real time while remaining open to new data being added?

The results in Table 1 show the response times to a set of queries performed on the full ACM metadata set of 700,000 papers, with the tests conducted on the three methods of data storage/retrieval described above: an open database queried by SQL, a closed heavily-indexed database queried by SQL and a 3Store Knowledge Base queried by SPARQL. The thirteen tests conducted were all based on queries that either the web services or the SemioViewer application need to ask at some point in their execution.

Table 1. Results of experiments performed on ACM dataset using SQL and SPARQL

Test #	Test	Closed SQL	Open SQL	SPARQL
1	Search for paper given incomplete title string (‘AKTive’).	0.83s	1.04s	23s
2	Search for author given incomplete name string (‘Keller’).	2.02s	2.03s	32s
3	Search for author given incomplete name string (‘Johnson’).	2.04s	2.04s	52s
4	Get paper details (title, year) given paper ID.	0.02s	1.28s	<0.5s
5	Get paper details (title, year, authors)given paper ID.	2.06s	3.40s	<0.5s
6	Get paper details (title, year, authors) given in-complete search string for title.	1m 34.97s	4m 0.566s	Times out.
7	Get paper’s top 15 cita-tions and relative impact scores (cite count, author-ity, combination) given paper ID.	1.27s	Times out (>30 mins)	Consis-tently <10s, typically <4s.
8	Get details of author given author ID.	0.03s	0.87s	~1.5s
9	Get all papers (paper ID only) by a particular au-thor given author ID.	0.01s	2.04s	~2s
10	Get all papers (paper ID, title, year) by a particular author given author ID.	0.78s	1m 4.03s	~3s
11	Get impacts of all papers from previous test and thus calculate author impact (author has 71 papers)	1.32s	~0.5s per individual query, 37s total; times out (>30 mins) if done as one query.	<15s
12	Get impacts of all papers from previous test and thus calculate author impact (author has 10 papers).	0.51s	~0.5s per individual query, 6s total; times out (>30	<5s

			mins) if done as one query.	
13	Get closest 15 co-authors and calculate their relative impact given author ID. (Tested on various author IDs).	Typically <10s. Maximum observed for complex author 19s.	Times out (>30 mins)	<15s for typical author. Maximum observed for complex author 27s.

The most noticeable results is the general speed advantage of the closed database: this significantly out-performs either the open database or the SPARQL triplestore queries in most cases. However realistically, while it is important for the purposes of fairness to MySQL to show the advantages of heavy indexing, the comparison in results that needs to be made is between the open SQL system and the SPARQL-based KB. There are four types of results reported in the above section: those where SQL on the open database was substantially faster than SPARQL (tests 1, 2, 3), those where open SQL and SPARQL were roughly the same and both usable (tests 4, 8, 9), those where SPARQL was substantially faster than open SQL (tests 5, 7, 10, 11, 12, 13) and those where neither open SQL nor SPARQL were quick enough to be useful (test 6). We now consider each type of result in turn.

Open SQL faster than SPARQL

These are queries dealing with questions of identity (single table information gathering): string matching within a particular field is something SQL is heavily optimised for, even without multiple indexing. SPARQL, conversely, does not contain a ‘LIKE’ function and instead relies on searching all records for subjects in triples with the predicate ‘has-title’ or ‘full-name’ and then filtering on a regular expression – a much more time-consuming process. Therefore for substring queries, typically searches, it is clear that SQL is superior and should be used in a practical, real-world system. Also note that test 8 produced a marginally quicker result for open SQL than SPARQL: this is due to an optimised RDBMS searching just a single table for more identity information about a given author ID, whereas the SPARQL query has to search through a few triples to get the required information.

Open SQL and SPARQL similar

These tended to be simple queries where open SQL had to only look in a single table and SPARQL had to only find a small number of predicate-subject combinations. In practice, either query type may be used for these queries and equally good results may be expected.

SPARQL faster than open SQL

The nature of the SemioViewer application means this is the largest group of results: calculating impacts and co-authorship communities requires a more intense study of relationships between data. For SPARQL, this is ideal: it has been optimised for searching object-predicate or predicate-subject combinations. For open SQL, the

Relational Database model allows for joins between tables but the queries quickly become complex (see examples in section 2, above) and for tables containing several hundred thousand rows, joins can be particularly time-consuming unless the multiple indexes in the closed system are applied. Tests 11 and 12 show that performing numerous individual queries rather than a single, more complex join operation can be more time-efficient, even if the end result is identical. However, this is programmatically more complex as it requires tailored scripts to be generated, and even then the SPARQL queries are generally quicker, particularly (as test 12 shows) for authors who have written a larger volume of papers.

Neither SPARQL nor SQL quick enough

This was a single, complex test which involved issues that lead to a struggle for both SPARQL (incomplete string querying) and open SQL (multi-table joins on large tables). Even the closed SQL system struggled with this: multiple indexing did not help with the 'like' query to the extent that it did with the other queries. In practice, the SemioViewer application breaks this query down into two stages: perform a search to get a paper ID (best performed using open SQL) and get the details of that paper and its authors (best performed using SPARQL). It is important to note that queries like this will exist when constructing applications for large-scale metadata stores, and the solution is to break it down into less complex queries and perform them sequentially using a suitable approach for each one.

Analysis

It is important to again point out that these experiments were performed on a single server using a particular instance of MySQL, on which the 3Store was built. The differences therefore can not be put down to superior hardware or database performance, but to the design differences between the RDBMS and RDF models of data representation. While it can be argued that the multiple indexing of the closed database provide better results for the SQL queries, this still leaves us with complex SQL statements performing JOIN operations on large tables, as well as the inherent problem of performing updates on what needs to be a live, frequently-updated system.

The results therefore show that in practice, the only realistic way for the SemioViewer application to work is to have both open SQL and SPARQL queries. While not typical Semantic Web applications, both the SemioViewer and the SPARQL-based web services and client pages described above require both SQL and SPARQL queries in order to perform effectively, if they are to remain open to having regular data updates. This is partly due to the design of SPARQL (certain features present in SQL are not included in SPARQL, such as there being no count(*) function and no 'like' facility within SPARQL) and partly due to nature of SPARQL and SQL: as suggested above, SQL is better at 'identity' queries, SPARQL superior at 'relationship' queries. With metadata for ~700,000 papers and 9 million triples, the only practical approach when creating live, updateable 'semiometrics' applications is to use both: open SQL for initial searching and SPARQL for getting more in-depth data for each paper or author, including information needed for influence analysis.

6 Conclusion

While the statistics produced of the SemioViewer application are interesting in themselves, the main conclusion to be drawn in this paper is that the RDF/SPARQL approach, along with a scalable triplestore solution, presents a viable alternative to SQL for large-scale metadata stores, particularly for queries based around relationship rather than identity. We have shown in this paper examples from a working application where SPARQL out-performs open SQL on both the theoretical and empirical level, as well as examples of SQL out-performing SPARQL. We have also shown that while a few simpler queries can be performed well using both approaches, there are very few that neither approach can handle in a reasonable time-frame: in these cases, simplifying queries provides the solution. In addition, we have shown that for systems that do not require frequent updates, a closed, heavily-indexed is preferable as it requires only one data source (an SQL database) rather than both a database and RDF KB; however, it has also been shown that for large-scale metadata stores requiring frequent updating, a closed system is impractical. It is therefore reasonable to conclude that when dealing with large-scale datasets featuring complex relationships and queries, RDF and SPARQL can provide a dramatically improved performance over the conventional RDBMS/SQL approach for certain queries.

Future work planned in this area includes looking at drawing two data sources together using the RDBMS and RDF approaches: for example, joining the ACM and Citeseer datasets currently held in distinct KBs, or perhaps the merging of different EPrints archives. While useful and theoretically simple in RDF (just put the resources in the same Knowledge Base) it raises the ongoing Semantic Web issue of co-reference resolution and duplicate records, and the question of whether RDBMS, RDF or both types of solution are necessary to solve this problem.

Acknowledgements

This work is supported under the Advanced Knowledge Technologies (AKT) Interdisciplinary Research Collaboration (IRC), which is sponsored by the UK Engineering and Physical Sciences Research Council under grant number GR/N15764/01.

We would like to thank Professor C. Lee Giles and Isaac Councill of Penn State University for the provision of Citeseer and for ongoing helpful comments.

References

1. Lawrence, S., Bollacker, K., Giles, C.L., "Digital Libraries and Autonomous Citation Indexing" IEEE Computer, 32(6), (1999). 67-71.
2. http://scholar.google.com
3. Lagoze, C., Van de Sompel, H.: The Open Archives Initiative: Building a low-barrier interoperability framework. Proceedings of the ACM/IEEE Joint Conference on Digital Libraries, Roanoke VA, USA (2001). 54-62.

4. Harris, S., Gibbins, N.: 3store: Efficient Bulk RDF Storage. Proceedings 1st International Workshop on Practical and Scalable Semantic Web Systems, Sanibel Island, Florida, USA. (2003)
5. Mc Bride, B.: Jena: Implementing the RDF Model and Syntax Specification, Proceedings of the Second International Workshop on the Semantic Web (2001).
6. The AKT Reference Ontology. http://www.aktors.org/publications/ontology/, (2002).
7. Weibel, S., The Dublin Core: A simple content description format for electronic resources. NFAIS Newsletter: 40(7), (1998). 117-119.
8. McRae-Spencer, D. M., Shadbolt, N. R., "Also By The Same Author: AKTive Author, A Citation-Graph Approach to Name Disambiguation", In Proceedings 6th ACM/IEEE Joint Conference on Digital Libraries, (2006), pp. 53-55.
9. McRae-Spencer, D. M., Shadbolt, N. R., "Semiometrics: Producing a Compositional View of Influence” (Preprint) (2006).