YARS2: A Federated Repository for Querying Graph Structured Data from the Web

Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker

National University of Ireland, Galway
Digital Enterprise Research Institute

Abstract. We present the architecture of an end-to-end semantic search engine that uses a graph data model to enable interactive query answering over structured and interlinked data collected from many disparate sources on the Web. In particular, we study distributed indexing methods for graph-structured data and parallel query evaluation methods on a cluster of computers. We evaluate the system on a dataset with 430 million statements collected from the Web, and provide scale-up experiments on 7 billion synthetically generated statements.

1 Introduction

The technological underpinnings of the Web are constantly evolving. With markup and representation languages, we have witnessed an upgrade from HTML to XML, mainly in the blogosphere where early adopters embraced the XML-based RSS (Really Simple Syndication) format to exchange news items. Data encoded in XML is better structured than HTML due to stricter syntax requirements and the tagging of data elements as opposed to document elements. Although the XML web is smaller in size than the HTML web, specialised search engines make use of the structured document content.

Whilst XML is appropriate in data transmission scenarios where actors agree on a fixed schema prior to document exchange, ad-hoc combination of data across seemingly unrelated domains rarely happens. Collecting data from multiple XML sources requires applications to merge data. The data merge problem is addressed by RDF, whereby, ideally, identifiers in the form of URIs are agreed-upon across many sources. In this scenario, RDF data on the Web organises into a large well-linked directed labelled graph that spans a large number of data sources.

There is an abundance of data on the Web hidden in relational databases, which represents a rich source of structured information that could automatically be published to the Web. Some weblog hosting sites have already begun exporting RDF user profiles in the Friend of a Friend (FOAF) vocabulary. Community-driven projects such as Wikipedia and Science Commons, and publicly funded projects – for example, in the cultural heritage domain – plan to make large amounts of structured information available under liberal licence models.

Hence, we see the benefit of a system that allows for interactive query answering and large-scale data analysis over the aggregated Web structured-data graph.

We study such a system as part of the Semantic Web Search Engine (SWSE) project. The goal of SWSE is to provide an end-to-end entity-centric system for collecting, indexing, querying, navigating, and mining graph-structured Web data. The system will provide improved search and browsing functionality over existing web search systems; returning answers instead of links, indexing and handling entity descriptions as opposed to documents. The core of SWSE is YARS2 (Yet Another RDF Store, Version 2), a distributed system for managing large amounts of graph-structured data.

Our work unifies experience from three related communities: information retrieval, databases, and distributed systems. We see our main contribution as identifying suitable well-understood techniques from traditional computer systems research, simplifying and combining these techniques to arrive at a scalable system to manage massive amounts of graph-structured data collected from the World Wide Web.

The remainder of this paper is organised as follows:

1. We describe the architecture and modus operandi of a distributed Web search and query engine operating over graph-structured data.
2. We present a general indexing framework for RDF, instantiated by a read-optimised, compressed index structure with near-constant access times with respect to index size.
3. We investigate different data placement techniques for distributing the index structure.
4. We present methods for parallel concurrent query processing over the distributed index.
5. We provide experimental measurements of scaling up the system to billions of statements.

2 Motivating Example

In the following we describe a scenario which current search engines fail to address: to answer structured queries over a dataset combined from multiple Web sources. A well interlinked graph-structured dataset furthermore enables new types of mining applications to detect common patterns and correlations on Web scale.

The use-case scenario is to find mutual acquaintances between two people. More specifically, the query is as follows: *give me a list of people known to both* `Tim Berners-Lee` *and* `Dave Beckett`. The query can be answered using data combined from a number of different sources.

Having aggregated all data from the sources, a query engine can evaluate the query over the combined graph. For our example query, `Dan Brickley` is one resulting answer to the question of *who are mutual acquaintances of Tim and Dave?*, that can only be derived by considering data integrated from a number of sources.

From the motivating example we can derive a number of requirements:

- **Keyword searches**. The query functionality has to provide means to determine the identifier of an entity[1] which can be found via keyword based searches (such as `tim berners lee`).
- **Joins**. To follow relationships between entities we require the ability to perform lookups on the graph structure. We cater for large result sets for high level queries, which is in contrast to Web searches where typically only the first few results are relevant.
- **Web data**. Since we collect data from the open Web environment, we need to pre-process the data (e.g., fusing identifiers); in addition, the index structures have to be domain independent to deal with schema-less data from the Web.
- **Scale**. Anticipating the growth of data on the Web, a centralised repository aggregating available structured content has to scale competently. The system has to exhibit linear scale-up to keep up with fast growth in data volume. A distributed architecture is imperative to meet scale requirements. To allow for good price/benefit ratio, we deploy the system on commodity hardware through use of a shared-nothing paradigm.
- **Speed**. Answers to interactive queries have to be returned promptly; fast response times are a major challenge as we potentially have to carry out numerous expensive joins over data sizes that exceed the storage capacity of one machine. To achieve adequate response times over large amounts of data, the indexing has to provide constant lookup times with respect to scale.

3 Preliminaries

Before describing the architecture and implementation of our system, we provide definitions for concepts used throughout the paper.

Definition 1. *(RDF Triple, RDF Node) Given a set of URI references $\mathcal{R}$, a set of blank nodes $\mathcal{B}$, and a set of literals $\mathcal{L}$, a triple (s, p, o) $\in (\mathcal{R} \cup \mathcal{B}) \times \mathcal{R} \times (\mathcal{R} \cup \mathcal{B} \cup \mathcal{L})$ is called an RDF triple.*

In a triple (s, p, o), s is called subject, p predicate or property, and o object. To be able to track the provenance of a triple in the aggregated graph, we introduce the notion of context.

Definition 2. *(Triple in Context) A pair (t, c) with a triple t and $c \in (\mathcal{R} \cup \mathcal{B})$ is called a triple in context c.*

Please note that we refer to a triple $((s, p, o), c)$ in context c as a quadruple or quad (s, p, o, c). The context of a quad denotes the URL of the data-source from hence the contained triple originated.

[1] e.g., `http://www.w3.org/People/Berners-Lee/card#i`

4 Architecture

We present the distributed architecture of SWSE, combining techniques from databases and information retrieval systems. A system orientated approach [6] is required for graph-based data from the Web because of scale. The system architecture of a Semantic Web Search Engine requires the following components:

- **Crawler**. To harvest web-documents, we use MultiCrawler [14]: a pipelined crawling architecture which is able to syntactically transform data from a variety of sources (e.g., HTML, XML) into RDF for easy integration into a Semantic Web system.
- **Indexer**. The Indexer provides a general framework for locally creating and managing inverted keyword indices and statement indices; we see these two index types as the fundamental building blocks of a more complex RDF index. Our framework, with combinations of keyword and statement indices, can be used to implement specialised systems for indexing RDF.
- **Object Consolidator**. Within RDF, URIs are used to uniquely identify entities. However, on the web, URIs may not be provided or may conflict for the same entities. We can improve the linkage of the data graph by resolving equivalent entities. For example, we can merge equivalent entities representing a particular person through having the same values for an email property; see [17] for more details.
- **Index Manager**. The Index Manager provides network access to the local indices, offering atomic lookup functionality over the local indices. Local indices can include keyword indices on text and statement indices such as quad indices on the graph structure, and join indices on recurring combinations of data values.
- **Query Processor**. The Query Processor creates and optimises the logical plan for answering both interactive browsing and structured queries. The Query Processor then executes the plans over the network in a parallel multi-threaded fashion, accessing the interfaces provided by the local Index Managers resident on the network.
- **Ranker**. To score importance and relevance of results during interactive exploration, we use ReConRank [16]. ReConRank is a links analysis technique which is used to simultaneously derive ranks of entities and data-sources. Ranking is an important addition to search and query interfaces and is used to prioritise presentation of more pertinent results.
- **User Interface**. To provide user-friendly search, query and browsing over the data indexed, we provide a user interface which is the human access point to the Semantic Web Search Engine. Users incrementally build queries to browse the data-graph – through paths of entity relationships – and retrieve information about entities.

The focus of the paper is on describing YARS2, the indexing and query processing functionality as illustrated in Figure 1. In the remainder of the paper, we first describe the Index Manager, next discuss the Indexer and data placement strategies, and then present the Query Processor.

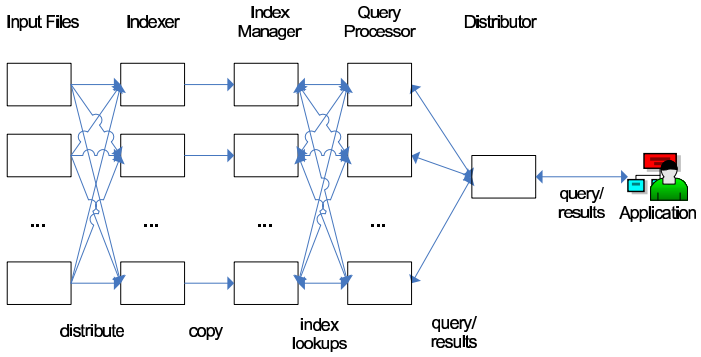

Fig. 1. Parallel index construction and query processing data flow.

5 Anatomy of the Index Manager

We require index support to provide acceptable performance for evaluating queries. The indices include

- a keyword index to enable keyword lookups.
- quad indices to perform atomic lookup operations on the graph structure
- join indices to speed up queries containing certain combinations of values, or paths in the graph.

For the keyword index, we deploy Apache Lucene[2], an inverted text index [20]. The keyword index maps terms occurring in an RDF object of a triple to the subject. We implement the quad index using a generic indexing framework using (key, value) pairs distributed over a set of machines. Similarly, join indices can be deployed using the generic indexing architecture. In the following, we illustrate the indexing framework using the quad index; join indices can be deployed analogously.

5.1 Complete Index on Quadruples

The atomic lookup construct posed to our index is a quadruple pattern.

Definition 3. *(Variable, Quadruple Pattern) Let $\mathcal{V}$ be the set of variables. A quadruple (s, p, o, c) $\in (\mathcal{R} \cup \mathcal{B} \cup \mathcal{V}) \times (\mathcal{R} \cup \mathcal{V}) \times (\mathcal{R} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{V}) \times (\mathcal{R} \cup \mathcal{B} \cup \mathcal{V})$ is called a quadruple pattern.*

A naïve index structure for RDF graph data with context would require four indices: on subject, predicate, object, and context. For a single quad pattern lookup containing more than one constant, such a naïve index structure needs

[2] `http://lucene.apache.org/java/docs/fileformats.html`

to execute a join over up to four indices to derive the answer. Performing joins on the quad pattern level would severely hamper performance.

Instead, we implement a complete index on quads [13] which allows for direct lookups on multiple dimensions without requiring joins. If we abstract each of the four elements of a quad pattern as being either a variable $\mathcal{V}$ or a constant $\mathcal{C} = \mathcal{R} \cup \mathcal{B} \cup \mathcal{L}$, we can determine that there are $2^4 = 16$ different quad lookup patterns for quadruples. Naïvely, we can state that 16 complete quad indices are required to service all possible quad patterns; however, assuming that prefix lookups are supported by the index, all 16 patterns can be covered by six alternately ordered indices. Prefix lookups allow the execution of a lookup with a partial key; in our case an incomplete quad.

We continue by examining three candidate data structures for providing complete coverage of the quad patterns. In examining possible implementations, we must also take into account the unique data distribution inherent in RDF. The most noteworthy example of skewed distribution of RDF data elements is that of `rdf:type` predicate; almost all entities described in RDF are typed. Also, specific schema properties can appear regularly in the data. Without special consideration for such data skew, performance of the index would be impacted.

5.2 Index Structure Candidates

For implementing a complete index on quadruples, we consider three index structures: B-tree, hash table, and sparse index[11].

- A **B-tree** index structure provides prefix lookups which would allow us to implement a complete index on quads with only six indices as justified in Section 5.1; one index can cover multiple access patterns. However, assuming a relatively large number of entries ($10^6 - 10^9$), the logarithmic search complexity requires prohibitively many disk I/O operations (20 - 30) given that we are limited as to the portion of the B-tree we can fit into main memory.
- **Hash-tables** enable search operations in constant time; however, a hash-table implementation does not allow for prefix lookups. A complete index on quads implemented using hash tables would thus require maintaining all 16 indices. The distribution of RDF data elements is inherently skewed; elements such as `rdf:type` would result in over-sized hash buckets. If the hash value of a key collides with such an oversized bucket, a linear scan over all entries in the hash bucket is prohibitively expensive.
- A third alternative, and the one we implement, is that of a **sparse index**, which is an in-memory data structure that refers to an on-disk sorted and blocked data file. The sparse index holds the first entry of each block of the data file with a pointer to the on-disk location of the respective block. To perform a lookup, we perform binary search on the sparse index in memory to determine the position of the block in the data file where the entry is located, if present. With the sparse index structure, we are guaranteed to use a minimum number of on-disk block accesses, and thus achieve constant lookup times similar to hash tables. Since the sparse index allows for prefix

lookups, we can use concatenated keys for implementing the complete index structure on quads.

5.3 Implementing a Complete Index on Quads

The overall index we implement comprises of an inverted text index and six individual blocked and sorted data files containing quads in six different combinations. For the sparse indices over the data files, we only store the first two elements of the first quad of each block to save memory at the expense of more data transfers for lookups keys with more than two dimensions.

More generally, the sparse index represents a trade-off decision: by using a smaller block size and thus more sparse index entries, we can speed up the lookup performance. By using a larger block size and thus less sparse index entries, we can store more entries in the data file relative to main memory at the expense of performance. The performance cost of larger block sizes is attributable to the increase of disk I/O for reading the larger blocks.

To save disk space for the on-disk indices, we compress the individual blocks using Huffman coding. Depending on the data values and the sorting order of the index, we achieve a compression rate of ≈ 90 %. Although compression has a marginal impact on performance, we deem that the benefits of saved disk space for large index files outweighs the slight performance dip.

Figure 2 shows the correspondence between block size and cumulated lookup time for 100k random lookups, and also shows the impact of Huffman coding on the lookup performance; block sizes are measured pre-compression. The average lookup time for a data file with 100k entries using a 64k block size is approximately 1.1 ms for the uncompressed and 1.4 ms for the compressed data file. For 90k random lookups over a 7 GB data file with 420 million synthetically generated triples, we achieve an average seek time of 8.5 ms.

6 Indexer and Data Placement

The Indexer component handles the local creation of the keyword and sparse indices for the given data. For our specific complete quad index, we require building six distinctly ordered, sorted and compressed files from the raw data. The following outlines the process for local index creation orchastrated by the Indexer component:

1. Block and compress the raw data into a data file ordered in subject, predicate, object, context order (SPOC).
2. Sort the SPOC data file using a multi-way merge-sort algorithm.
3. Reorder SPOC to POCS and sort the POCS data file.
4. Complete step 3 for the other four index files.
5. Create the inverted text index from the sorted SPOC index file.

We performed an initial evaluation of the multi-way merge-sort of a file containing over 490M quads. We sorted segments of the file in memory, wrote the

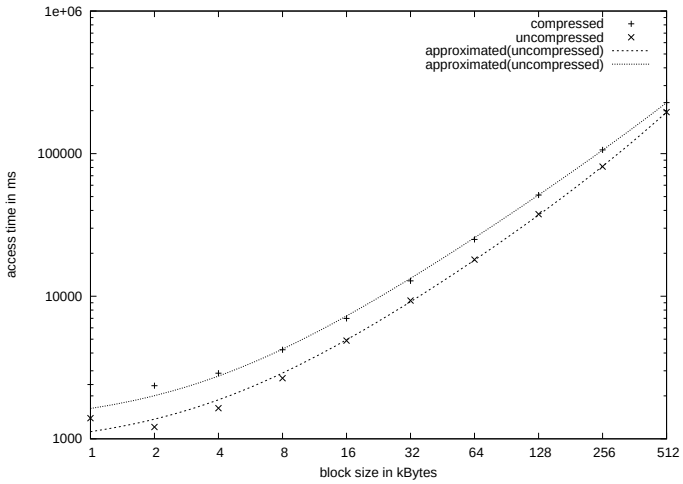

Fig. 2. Effect of block size on lookup performance using uncompressed and compressed blocks. We performed random lookups on all keys in a file containing 100k entries with varying block sizes. Results plotted on log/log scale.

sorted quads to batch files, and then merge-sorted the resulting batch files. Depending on the size of the in-memory segments, the process took between 19 hours 40 minutes (80k statements in-memory) and 9 hours 26 minutes (320k statements in-memory).

Thus far, we have covered local index management. Since our index needs to implement a distributed architecture for scalability and we require multiple machines running local Index Managers, we need to examine appropriate data placement strategies.

We consider three partitioning methods to decide which machine(s) a given quad will be indexed on:

1. random placement with flooding of queries to all machines
2. placement based on a hash function with directed lookup to machines where quads are located
3. range-based placement with directed lookups via a global data structure

We focus on the hash-based placement, which requires only a globally known hash function to decide where to locate the entry. The hash placement method can utilise established distributed hash table substrates to add replication and fail safety. For more on how to distribute triples in such a network see [8].

We avoid complex algorithms to facilitate speed optimisation. The peer to which an index *entry* (e.g. SPOC, POCS) is placed is determined by:

$peer(entry) = h(entry[0]) \mod m$

where m is the number of available Index Managers.

Hashing the first element of an index entry assumes an even distribution of values for the element which is not true for predicates. The issue of load balancing based on query forwarding in hash-distributed RDF stores has been investigated in [2]. However, a simpler solution which does not require query forwarding is to resort to random distribution where necessary (for POCS), where the index is split into even sizes, and queries are flooded to all machines in parallel.

To evaluate the indexing component, we created a univ(50000) dataset using the Lehigh University Benchmark [12], which we adapted to also produce variable-length text strings from an English dictionary in order to test Lucene. Table 1 summarises the indices deployed for the scale-up experiments.

Description	1 Machine	16 Machines
Number of statements	425 million	6.8 billion
Index size (complete index)	6*7 GB	672 GB
Index size (lucene)	16 GB	256 GB

Table 1. Index statistics for syntectically generated dataset.

7 Distributed Query Evaluation

We implement a general-purpose query processor operating on multiple remote Index Managers to enable evaluation of queries in SPARQL format[3]. In this section, we

- discuss network lookup optimisations for stream-processing large result sizes and evaluate our approach with a dataset of 7 billion statements
- devise a query processing method to perform joins over the distributed Index Managers.

7.1 Atomic Lookups over the Network

Before we can perform join processing in the Query Processor, we must implement optimised methods for handling the network traffic and memory overhead involved in sending large amounts of atomic lookup requests and receiving large amounts of response data over the network, to and from the remote Index Managers.

We implement multi-threaded requests and responses between the Query Processor and the Index Managers. For example, with our flooding distribution, each machine in the network receives and processes the lookup requests in parallel.

To be able to handle large result sets, we have to be careful not to overload main memory with intermediate results that occur during the query processing

[3] `http://www.w3.org/TR/rdf-sparql-query/`

and therefore we need a streaming results model where the main memory requirements of the machines are finite since results are materialised in-memory as they are being consumed.

For a quad pattern lookup, multiple remote Index Managers are probed in parallel using multiple threads. The threaded connections to the Index Managers output results into a coordinating blocking queue with fixed capacity. The multiple threads synchronise on the queue and pause output if the queue capacity is reached.

Iterators that return sets instead of tuples to increase performance have been described in [18] as row blocking. We measure the impact of row blocking via an index scan query over 2, 4, 8, and 16 Index Managers. Each index manager provides access to a over 7 GB data file with 420 million synthetically generated triples, which amounts to a total capacity of roughly 7 billion statements. To be able to test keyword performance, we changed the string values in the Lehigh benchmark to include keywords randomly selected from a dictionary.

Figure 3 shows the impact of varying row blocking buffer size on the network throughput. As can be seen, throughput remains constant despite increasing the number of Index Managers servicing the index scan query. From this we can conclude that a bottleneck exists in the machine consuming results.

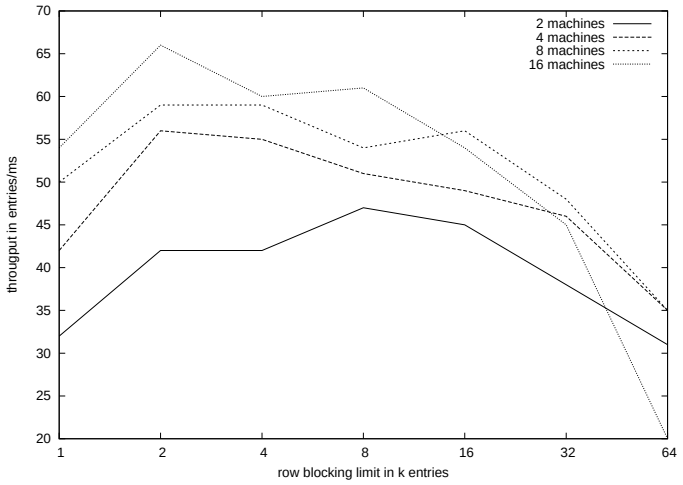

Fig. 3. Throughput for index scan with varying row blocking sizes.

7.2 Join Processing

We begin our discussion of join processing by introducing the notions of variable bindings, join conditions, and join evaluation and continue by detailing our method of servicing queries which contain joins.

Definition 4. *(Variable bindings) A variable binding is a function from the set of variables $\mathcal{V}$ to the set of URI references $\mathcal{R}$, blank nodes $\mathcal{B}$, or literals $\mathcal{L}$.*

Definition 5. *(Join Condition) Given multiple quad patterns in a query, a join condition exists between two quad patterns $\mathcal{Q}_j$ and $\mathcal{Q}_k$ iff there exists one variable $v \in \mathcal{V}, v \in \mathcal{Q}_j, v \in \mathcal{Q}_k$. Joins are commutative. Variable v is termed the join variable.*

In our query processing system, a query may consist either of one quad pattern (an atomic lookup) or may consist of multiple quad patterns where each pattern satisfies the join condition with at least one other pattern.

For joins we use a method called index nested loops join [11]. Multiple join operations can run concurrently in individual threads, with queues as coordination data structures for data exchange between the operators. Figure 4 illustrates the parallel execution of joins across remote Index Managers coordinated by the main thread M. Queues are represented as stack of boxes. Thread S represents a lookup operations of the first quad pattern in a query. The lookup is flooded to n Index Managers via threads $S_{1...n}$. The alternative would be to perform a directed lookup via the hash function. Intermediate results are passed to the join thread J, which in turn floods the lookups to n Index Managers via threads $J_{1...n}$. Threads $J_{1...n}$ write final join evaluations to a blocking queue, which is accessed by the main thread M.

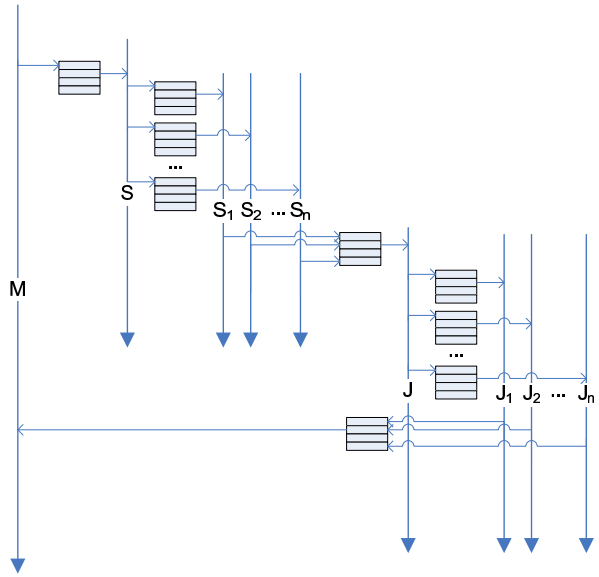

Fig. 4. Concurrent query execution with threads for exchanging intermediate results.

A necessary optimisation for joins requires that we carefully select which quad pattern will be serviced first to find initial valuations for the join variable. For join reordering, we can utilise a dynamic programming approach.

To evaluate the performance of distributed join processing, we deployed the 7 billion dataset over 16 Index Managers on 16 machines, and put the query processing component on a 17th machine. We tested 100 queries with a randomly chosen resource joined with one or two quad patterns. Figure 5 illustrates the correspondence between performance and result size.

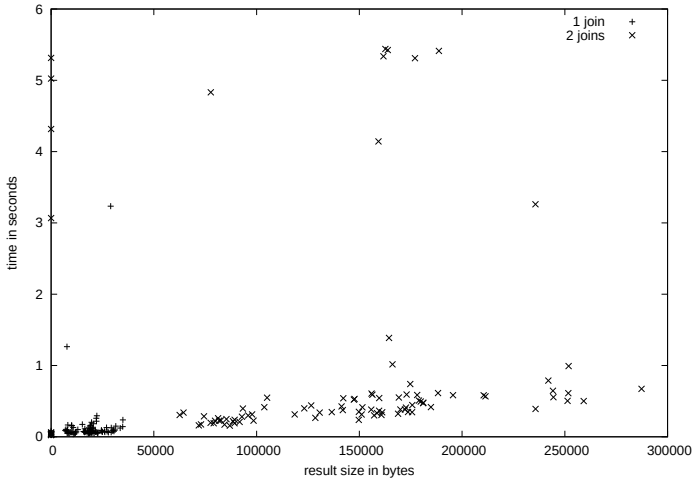

Fig. 5. Distributed join evaluation performance depending on the result size

8 Related Work

We employ variations of well-understood techniques from the fields of information retrieval, databases, and distributed systems. Inverted indices are discussed in Salton and McGill [20]. Our sparse index implementation for quads and supporting indices can be seen as a BTree index [3] with height 2, where the first level is entirely kept in memory. We optionally use compression, whose importance is well motivated by [23]. The idea of using multiple sorting order for keys to allow multidimensional lookups stems from [19]. Kowari [24] uses a similar complete quadruple index implemented using a hybrid of AVL trees and B-Trees. Semijoins, a method for performing joins in distributed databases has been introduced by Bernstein and Goodman [4]. Selinger et al. [21] introduced dynamic programming as a method for deriving query plans.

The WebBase [15] project describes in detail the architecture of a medium-sized Web repository, and various choices for implementing such a system. In

contrast to documents, we deal with structured data. Swoogle [9] uses information retrieval methods to provide keyword searches over RDF data on a single machine. In contrast, we provide structured query processing capabilities on a distributed architecture.

Sesame [7] is one of the early RDF stores operating on one machine. Cai and Frank [8] propose a method to distributed RDF storage on a distributed hash table substrate. Stuckenschmidt et al. [22] investigate theoretically the use of global indices for distributed query processing for RDF. A treatment of RDF from a graph database perspective can be found in [1]. We have made a step towards unifying query processing with Web search; adding reasoning functionality to the mix [10] is the next step.

9 Conclusion

We have presented the architecture of a federated graph-structured data repository for use in a Semantic Web search engine, described various implementation alternatives, and provided experimental and theoretical performance evaluation of the parallel system. To handle the complexity of a system involving a large number of machines, and to be able to optimise the performance of the individual operations, our data structures and methods have to exhibit good scale-up properties. We thus devised local data structures with constant seeks and linear throughput, optimised network data transfer, and multi-threaded query processing to achieve acceptable query performance on large data sets in a federated system.

Acknowledgements

This work has been supported by Science Foundation Ireland under project Lion (SFI/02/CE1/I131) and by the European Commission under project TripCom (IST-4-0027324-STP).

References

1. R. Angles and C. Gutiérrez. Querying rdf data from a graph database perspective. In *Proceedings of the Second European Semantic Web Conference*, pages 346–360, 2005.
2. D. Battré, F. Heine, A. Höing, and O. Kao. Load-balancing in p2p based rdf stores. In *2nd Workshop on Scalable Semantic Web Knowledge Base System*, 2006.
3. R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.
4. P. A. Bernstein and N. Goodman. Power of natural semijoins. *SIAM Journal on Computing*, 10(4):751–771, 1981.
5. P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*, pages 225–237, 2005.

6. E. A. Brewer. Combining Systems and Databases: A Search Engine Retrospective . *Readings in Database Systems, 4th. Edition*, 1998.
7. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the 2nd International Semantic Web Conference*, pages 54–68. Springer, 2002.
8. M. Cai and M. Frank. Rdfpeers: a scalable distributed rdf repository based on a structured peer-to-peer network. In *Proceedings of the 13th International World Wide Web Conference*, pages 650–657. ACM Press, 2004.
9. L. Ding, T. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. C. Doshi, and J. Sachs. Swoogle: A Search and Metadata Engine for the Semantic Web. In *Proceedings of the Thirteenth ACM Conference on Information and Knowledge Management*. ACM Press, 2004.
10. D. Fensel and F. van Harmelen. Unifying reasoning and search to web scale. *IEEE Internet Computing*, 11(2):96, 94–95, 2007.
11. H. Garcia-Molina, J. Widom, and J. D. Ullman. *Database System Implementation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.
12. Y. Guo, Z. Pan, and J. Heflin. An Evaluation of Knowledge Base Systems for Large OWL Datasets. In *Proceedings of the 3rd International Semantic Web Conference*, pages 274–288. Springer, 2004.
13. A. Harth and S. Decker. Optimized index structures for querying rdf from the web. In *Proceedings of the 3rd Latin American Web Congress*, pages 71–80. IEEE Press, 2005.
14. A. Harth, J. Umbrich, and S. Decker. Multicrawler: A pipelined architecture for crawling and indexing semantic web data. In *Proceedings of the 5th International Semantic Web Conference*, pages 258–271, 2006.
15. J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke. WebBase: a repository of Web pages. *Computer Networks*, 33(1–6):277–293, 2000.
16. A. Hogan, A. Harth, and S. Decker. ReConRank: A Scalable Ranking Method for Semantic Web Data with Context. In *2nd Workshop on Scalable Semantic Web Knowledge Base Systems*, 2006.
17. A. Hogan, A. Harth, and S. Decker. Performing object consolidation on the semantic web data graph. In *Proceedings of 1st I3: Identity, Identifiers, Identification Workshop*, 2007.
18. D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.
19. V. Y. Lum. Multi-attribute retrieval with combined indexes. *Communications of the ACM*, 13(11):660–665, 1970.
20. G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill Book Company, 1984.
21. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 International Conference on Management of Data*, pages 23–34, 1979.
22. H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, and J. Broekstra. Index Structures and Algorithms for Querying Distributed RDF Repositories. In *Proceedings of the 13th International World Wide Web Conference*, pages 631–639, 2004.
23. I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.
24. D. Wood, P. Gearon, and T. Adams. Kowari: A platform for semantic web storage and analysis. In *XTech 2005 Conference*, 2005.