String-Based Semantic Web Data Management Using Ternary B-Trees

Jürg Senn
Department of Computer Science
University of Basel
juerg.senn@unibas.ch

Supervised by: Helmar Burkhart
Department of Computer Science
University of Basel
helmar.burkhart@unibas.ch

ABSTRACT

The Resource Description Framework (RDF) stems from the Semantic Web but can also be regarded simply as a data model, independent of its origins. Its simple structure is ideal for describing and merging heterogeneous data from different sources quickly, without having to design a complex schema first. The different nature of RDF requires new approaches for data management and query processing and entails new problems when it comes to efficiency and scalability. Current solutions rely on string mapping and extensive indexing, but concentrate on a subset of the RDF query language SPARQL. They also ignore inherent properties residing in RDF itself, particularly its string characteristics.

In this work we propose the ternary B-tree as a new data structure for storing and accessing RDF. It is string-based, making use of the intrinsic features of RDF. Strings are decomposed into fixed-size elements and organized as tree nodes. This approach permits new ways of filtering RDF, joining query results, partitioning for different orderings, and for parallelization.

1. INTRODUCTION

The Resource Description Framework (RDF, [5]) is a simple data model for representing statements or facts in a knowledge base. It originates from the Semantic Web field where it constitutes the basic building block to model knowledge in a structured form. In the context of the Semantic Web, RDF is the means to convey knowledge related data worldwide in an unambiguous manner. Its fully defined structure is machine-readable, in contrast to the semi-structured information published on the Web today. And most importantly, it provides the elementary facts for automatic logical reasoning, the cornerstone of the Semantic Web.

It has since been recognized that RDF as a data model is applicable in a more general way. Its simple format is ideal for collecting and storing any kind of data quickly, without the need to create a complex schema first. Compared to the relational model, we could say that the intended meaning of data is provided implicitly when accessing it, not when designing the schema. Having no schema also simplifies amendment and extension. New RDF statements can be added immediately and provide further information about elements described. All in all, RDF is seen from a data model perspective this way, in contrast to the logics perspective mentioned above. This observation of RDF taking multiple roles is also made in [11]. Examples abound where the data model perspective is taken and information is currently published foremost with eased reuse in mind, not logical reasoning. In particular, the Linked Data initiative [2] has become something of a hub, compiling a directory of datasets and encouraging interlinking between them. Concrete examples of freely published data are the protein sequence database Uniprot [38], structured information extracted from Wikipedia [1, 37], U.S. census data [6], geographical data [7], and reviews from the Revyu web site [3], only to mention a few.

The top portion of Figure 1 provides an example of facts stated in RDF, taken from [33]. Each fact is represented by a triple where the first element is the subject to be described, the second denotes a property, and the third a value for the property (the object). The elements of the triple are called resources in RDF parlance. In the example, books are described with their title. The values of the first two books are literals whereas the third is a URI identifying another resource. The latter could act as a subject of other triples again, introducing a third perspective of RDF which interprets it as a graph structure. The middle portion of Figure 1 is an example of the query language for RDF, called SPARQL [33]. Books are selected with a triple pattern and a regular expression, then ordered by title. SPARQL queries always require one or more triple patterns matching a portion of the dataset. Each pattern match returns a set of variable bindings which have to be joined by common variables. The binding sets are filtered, ordered, and projected to yield the final list of results.

The simplicity and ease of use of RDF is countered with its fine-grained nature. Queries typically require a large number of patterns resulting in complex query graphs. During query optimization, more alternatives must be considered. In terms of physical design, RDF requires new approaches for compact storing and efficient access. RDF datasets tend to grow fast which poses scalability problems. These challenges have already been approached with considerable success, but several important issues require further consideration.

VLDB 2010 PhD Workshop, September 13, 2010, Singapore

```
@prefix :   <http://exmpl.org/book/>.
@prefix dc: <http://purl.org/dc/elements/1.1/>.
@prefix title: <http://exmpl.org/book/title/>.

:book0 dc:title "Web Handbook" .
:book1 dc:title "SPARQL Tutorial" .
:book2 dc:title title:The_Semantic_Web .
```

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?title
WHERE { ?x dc:title ?title
        FILTER REGEX(?title, "Web") }
ORDER BY ?title
```

```
?title = <http://exmpl.org/book/title/
          The_Semantic_Web>
?title = "Web Handbook"
```

Figure 1: RDF Example, SPARQL Query, and Query Result

2. RELATED WORK

Query engines supporting RDF and SPARQL can be categorized coarsely by their type of physical design. An early system is RDFSuite [10] which maps to an object-relational database. The schema for the mapping is still heavily based on semantic properties of the underlying data and uses RDF Schema (now part of [5]). Sesame [18] also builds relational tables from the data semantics but has an abstraction layer which permits use of different database systems. 3store [22, 23] saves RDF to a single, giant triples table with columns for three resources and uses a relational database as the back-end. Resources are hashed to integer identifiers and kept in separate dictionary tables. Note that separation of structure and original resource strings by mapping to fixed-length values is universal across all references cited here. The reason is to avoid redundant storage, to compress data, and to have faster processing.

Jena [42, 19] is one of the most complete systems available. It implements most of the different Semantic Web specifications. Query processing is mainly memory-based but data can be stored in relational databases through a persistence layer. Jena originally had a layout similar to 3store. To avoid problems having many self-joins on a single triples table, the schema used now is denormalized and parts of the resource strings are written from the dictionaries back to the triples table. In addition, data is clustered according to common property resources. Triples having the same properties are written to different tables where the columns represent the properties, the rows subject and object(s). A similar approach is [20]. Continuing on the idea of building different property tables, the authors in [8, 9] suggest to vertically partition data. The idea is to create a separate relation for each property in the dataset. A relation has only two columns for subject and object. This approach avoids sparse tables with many NULL values and eases the use of multi-valued attributes. Clustering properties together requires extensive tuning to get good performance whereas partitioning vertically produces a schema automatically. It is also shown that this approach is well-suited for column stores. A rebuttal challenges the claims [35]. It is pointed out that there are scalability problems if there are many properties each requiring a separate table. Additionally, queries containing triple patterns where the property is a variable will require scanning all tables, making the approach non-universal.

YARS [24] and Kowari [43] introduce the idea of using index structures to provide access with different patterns occurring in queries. They both suggest indexes built with the different possible permutations of the triple elements (Subject Property Object (SPO), SOP, PSO, . . .), but they only suggest the cyclic orders SPO, POS, and OSP, missing the need for other orders required in complex queries. YARS implements the indexes with B-trees, Kowari uses a mix between B-trees and AVL trees. Solutions that exhaustively index all possible orderings are RDF-3X [31] and Hexastore [41]. These approaches forego the need for a relational schema and a mapping to it. The indexes themselves become the database. Experiments in [31] and [32] demonstrate a vast improvement over previous approaches in terms of query performance, given the right query optimization. Jena now also has index storage as an option with TDB [4].

Relational database mappings and approaches using indexes are the most common types of RDF engines. Other types come in different classes. They can be graph-based [17], distributed [39], bitmap-based [12], or vector-based [40, 26].

Query optimization is not always addressed thoroughly in approaches that use relational databases as their back-end. It is commonly assumed that the optimizer of the database solves this task. Early works on SPARQL query optimization adopt known techniques like gathering statistics [36] or use a more sophisticated cardinality estimation [29]. The optimizer of RDF-3X [31] uses dynamic programming and precomputes frequent paths. It is extended in [32] with more accurate selectivity estimation. The OneQL [27] system employs a hybrid strategy combining cost-based and magic set optimization techniques.

3. PROBLEM STATEMENT

Index-based solutions for RDF engines demonstrate that adaptation of earlier methods to RDF specific specializations results in much more efficient and scalable systems. But currently, most results of research RDF engines are based on experiments with queries that consist almost exclusively of a single triple pattern set. We see a problem when extending the engines to queries that require extensive filtering and ordering. The issue with filters is recognized in [28] and [40], but the former gives a solution by mapping to a single SQL query and does not use special indexes, the latter does not address the problem.

Filters may contain terms that need to operate on the original string elements of triples. The filter in Figure 1 is an example where a regular expression is applied to an object of a triple. To execute this filter, an engine that normally works with fixed-length integer representatives must fetch the associated string from its separate dictionary. Queries that have triple patterns with low selectivity but contain highly selective filters will suffer a drop in performance. The same is true for lexicographic comparison of strings with filters. One could add some type information to the integer values. But this only helps with a specific subset of filters, not with string filters.

Many queries will also require an ordering of results. But the mapping from strings to integers might not be order-preserving. Even worse, there does not seem to exist a practicable way for mapping between general strings and a fixed integer set that retains the order. A theoretic solution is given in [25], but has issues with precision in practice. In other work the problem is acknowledged, but postponed [15]. Ordering might not be an issue if a query yields few results. But again, access to the dictionary may hurt execution times if there are many results. Concerning orders, partitioning an index into parts is also more problematic. Partitioned data might be interesting for parallel processing or sorting in memory. But partitioning should be done on the original strings, not on their identifiers. Otherwise, they are not ordered in a useful way (there could still be mapped data inside a partition, though).

Another aspect with string mapping is that inherent properties of the original data are not considered at all. Most of the resources in RDF triples are URIs (literals seem to be rarer). URIs have a hierarchical structure which can be exploited, e.g. when indexing or partitioning data. Use of properties like this requires working with the strings.

To extend existing solutions further, a more string-based approach should be considered. We believe that queries based on strings are quite common. For example, as RDF does not have a schema, the structure of an unknown database first has to be discovered. If the only way to do this is a SPARQL query, one must rely on filters.

4. APPROACH

In this thesis we propose an alternative approach for an RDF engine compared to established methods. We want to focus on a string-based data structure and processing method. The idea is to work with original, unmapped RDF data exclusively, to exploit its intrinsic properties for data storage, query optimization, query execution, and parallelization. We want to contribute:

- a novel disk-based string index dubbed *ternary B-tree*
- a compression scheme for this index to save space while still having fast read access
- an adapted query optimizer and processor which makes use of the index and RDF properties
- a partitioning scheme to tap the potential of parallel processing and to provide more interesting orders for the optimizer to consider

Our goal is to demonstrate that a string-based approach for an RDF engine can be viable and compete with previous solutions, despite the fact that we work with variable-sized data. However, we want to concentrate on read-mostly usage scenarios. Fast updates are not a primary concern. Currently, RDF datasets often come in bulks and are not changed frequently. Thus querying is more important at the moment.

In the following, we present our proposed index data structure as a first preliminary result and give some suggestions on how to make use of its properties. Due to space constraints, we will only be able to give an impression of the overall idea but cannot go into detail.

4.1 Ternary B-Trees

A ternary B-tree is an amalgamation of two data structures, the (generalized) ternary search tree [14] and the (prefix) B-tree [13]. When triples are inserted into a ternary B-tree, they first have their three parts concatenated into one string according to the designated order of the index (e.g. Property Subject Object (PSO)). This single string is then split up into elements of equal size. We currently use eight UTF-8 characters. Each element is then 64 bits wide. If a part is not a multiple of the element size, it is filled up with zeroes. This way we can determine if we are switching from one of the three parts to another. Combined elements from several triples form a multi-way tree with elements of common prefixes shared. This is illustrated in Figure 2 which contains elements from five triples arranged in PSO order. The different colors of the nodes point out the property, subject, and object of each triple, respectively. The tree is reminiscent of a Patricia trie [30], but it has fixed-length character sequences per node.

The tree can also be seen as a generalization of a ternary search tree [14]. We use the term ternary not because of structural aspects (nodes can have more than three children), but because of the way the tree is traversed. Tree iterators matching triple patterns follow nodes as long as they are equal to the known elements of the pattern. If the current element of the pattern searched is smaller or greater than the one found, the iterator immediately stops matching and follows the remaining elements with the closest match until the end is reached. The last element of each triple is a special terminator which holds the index of the next node to search.

Triples are organized into ternary search trees and the trees in turn form nodes of a B-tree. The B-tree nodes have a fixed size (e.g. 16 KB) and are split whenever the internal ternary search trees do not fit. The elements of a ternary search tree are stored breadth-first in the B-tree node including some directory information that describes the internal tree structure. With the directory, a B-tree node can then always hold any internal tree up to a certain maximum number of elements, regardless of structure.

The B-tree nodes themselves are stored sequentially in files, clustered according to their parent nodes. For example, the tree in Figure 2 has five terminators. If the tree is inside a B-tree node and that node is internal in the B-tree, it has six children that are other B-tree nodes (there is an implicit index 0). Those six nodes are stored together in one file. The terminator indicates which node index in the file has to be accessed. With this scheme, the nodes can be compacted into fewer files. The more triples that fit into a B-tree node, the less files there are.

The design of the ternary B-tree allows us to exploit the structure of RDF triples. A pattern match corresponds to a traversal of the tree. We do not have to read the triple strings in full because we split them up into elements and an element can be read fast if it is interpreted as a 64-bit integer. A join can be executed by matching prefixes of the triples. If the prefix of a subtree does not match, the whole subtree can be discarded. With this strategy we can exploit the string-based nature of a ternary B-tree.

4.1.1 *Index Building*

A ternary B-tree is built by inserting new triples sequentially. First, the appropriate leaf node for the triple is found. If the triple is not already in one of the nodes visited during traversal, it is added to the leaf. Adding a new triple to a node requires rebuilding the ternary subtree therein. The subtree is scanned breadth-first. In each level scanned, the appropriate position for the next element of the triple is de-

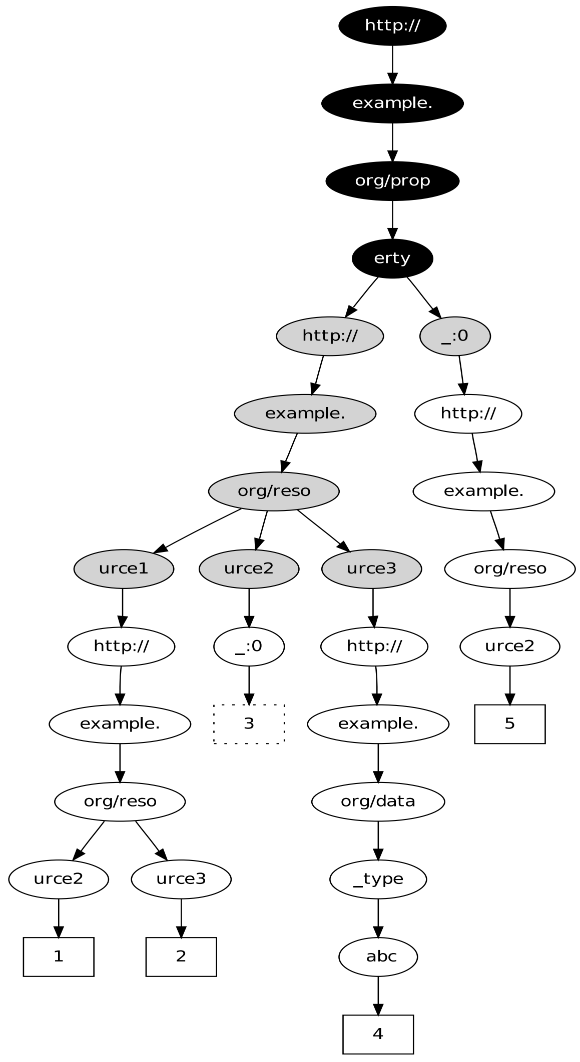

Figure 2: Example of a multi-way search tree containing five triples, in PSO order

termined. During the scan, all the existing elements of the tree are written. Once the position for the new element in one level is reached, it is added and the scan continues. Once all elements including the new ones are written, the insert is completed with the subtree containing the new triple.

If the number of elements in the new subtree exceeds the maximum number of elements, the node is split. The split algorithm selects one triple which is moved to the parent node. The trees formed from the elements on the left and right of the selected triple are written into two nodes. The algorithm selects the triple which most evenly distributes the elements in two, taking into account the common prefixes that are replicated in both halves. In Figure 2, this is the triple with terminator 3 (dotted). Moving one triple to the parent node, the split algorithm continues recursively until no more split is required or the root is reached. If a parent node is split, the file containing its children is also split.

Note that the split causes a full triple to be moved to the parent node. This triple is not part of the two split nodes anymore which means that in contrast to the prefix B-tree, strings in internal nodes of ternary B-trees are not only forks, but also actual data. The advantage of this is that we do not replicate prefix strings but on the other hand we cannot restrict ourselves to leaf nodes when scanning.

4.2 Partitioning

Materializing all possible permutations, we create six indexes (SPO, SOP, PSO, POS, OSP, OPS). In addition, we build accumulated indexes SP, SO, PS, PO, OS, OP, and S, P, O. This is suggested in [31] to use for patterns where a variable is not part of the result. The additional indexes also act as histograms for selectivity estimation. The main problem with keeping all possible permutations is the increase in storage. But due to their regular patterns, the string indexes are highly compressible (see our suggestions in Section 5.1).

We propose an additional set of indexes formed from the leaf nodes of a ternary B-tree. Recall from the previous section that tree nodes are stored sequentially in a file. The number of nodes in a file depends on the number of triples kept in the parent nodes. The files that contain the leaf nodes form the bottom layer of the tree. Let us now assume the generic query shown in Figure 3.

```
SELECT ?c ?b
WHERE { <s1> <p1> ?b .
        ?b   <p2> ?c }
ORDER BY ?c ?b
```

Figure 3: Generic Query with Order Requirement

This query has two patterns which join on ?b. One way to process this query is to use the PSO index for both patterns which yields variable bindings ordered by ?b first, then ?c. To get the final result, we have to reorder. This is no problem for highly selective queries with few results but if there are many results or the patterns are part of a bigger query with more complex order requirements, large variable binding lists have to be ordered and there are fewer chances to use pipelining. To gain more flexibility, we suggest replicating the leaf files and create additional, small indexes from them that are ordered differently. For example, if we take the POS index, the leaf files are ordered by P first, then O and S. We copy the leaf files and create indexes from them in order PSO. Triples that were moved to higher levels of the B-tree during splits are also copied to the appropriate indexes, as if they were still in the leafs. The new indexes are partitions in POS order but can be searched themselves in PSO order. The query of Figure 3 can then be answered in a different way. The first pattern is evaluated with the PSO index as before, but the second now uses POS first. If we reach a leaf file, we switch to the corresponding PSO index and join its results with the ones from the first pattern. This is repeated for each leaf file encountered. In other words, we join each partition separately. If there are only few results per partition, this will require many more disk reads compared to the normal approach, but it has the advantage that the result parts are already partitioned in the final order and can be sorted in isolation. In addition, if the leaf files are small enough, results can often be ordered in memory, even for large databases. This means that almost every query, no matter how complex, can eventually be answered by this scheme, without running out of resources.

Adding reordered indexes to the database gives us more flexibility for query processing but it is not yet known if the additional space requirements will be outweighed by the performance gains and if there are enough scenarios where using the indexes is actually faster than the traditional approach. It also adds more complexity to the query optimizer. But the potential gains should not be ignored and could give more significance to using RDF over the relational model. Exhaustively indexing orderings is simply not possible with the latter in all cases but conceivable with a triple model.

5. FUTURE WORK

In this section we list the work we have planned for the continuation of the project. Although it comprises the major part of our proposed contributions, we do not feel that the points mentioned here have matured enough to be included in the approach section and therefore mark them as future work.

5.1 Index Compression

Our current incarnation of the string index does not include any compression apart from combining common prefixes inside a node. We see two possibilities to reduce the overall index size.

The first is compression of the nodes of each file after some bulk insertion or after a certain time period (e.g. once per day). Each node is compressed individually. Up until now, we assumed that a file has a regular pattern where each node has the same size. This allows us to address a node immediately by the index given in the parent node. After compression, the nodes will be of different size. This requires a directory with a prefix sum array to get the correct position and size of the node in the file indirectly. The directory can be stored in a separate block at the beginning of the file or included in the terminator elements of the parent node.

The second possibility for compression is after each insert of a triple. The idea is to keep the nodes at their regular size and pack as many triples into them as possible. If the triples take too much room, even in their compressed state, the node is split. In comparison to the first method, compressing the node immediately has the advantage of delaying splits as long as possible which should improve insertion speed. On the other hand, one has to guarantee that the compressed node can be split into exactly two new nodes. A split into more than two nodes might be possible but is more complicated to handle. What is preferred then is a method to determine if the node must be split for a new triple or if the triple can still be included with a future split fitting into two nodes.

5.2 Optimizer

A query optimizer adapted to our string index is a core requirement and has to be the main contribution. We want to build on work presented in [31, 32, 27] and create our own optimizer. A major problem is to determine when to use the partitions suggested in Section 4.2 and when to use the original index. Another is how to gather and store statistics and how to apply them for the tree structure.

It is very likely that the optimizer has to contain a dynamic component which determines during query processing where filtering is done, how early in the process filtering is started, and if filtering is done for all elements passed during tree traversal or only intermittently. As the triples are stored in fixed-size elements, they can often be filtered out without having to read them completely. Continuing this thought, we can often exclude whole subtrees without having to traverse them in full which could help tremendously with joins.

An open question is if the additional partitions will allow us to produce the final requested ordering of data immediately without having to fall back to intermediate sorting. If this is possible, all joins can be merge joins. There would be no need for hash joins or similar pipeline-breaking operations. But it seems unlikely, as queries with high selectivity will have their results spread out into the partitions and reading the data would be much slower than using the original index with the wrong order and sorting afterwards. The optimizer will have to determine when to use which option.

5.3 Parallel Processing

Parallel processing of RDF data has not been studied at all up until now and we see opportunities to include this in our work. There are distributed solutions, but we want to concentrate on multi-core parallelism.

Having partitioned data provides the opportunity to process query operations in parallel, although it is difficult to assess the potential performance gain as access to disk is still serial. A join is separated into independent parts and threads can produce and cache results independently, at least if they are not in the same partition. As the indexes are ordered by the original triple strings, the different result parts will also be partitioned in that order and sorting is only required inside a partition.

An open question is on how to traverse the index trees in parallel. Is a new thread spawned for each fork or do the threads traverse the tree by the same path until the final, distinctive elements of the triples are reached?

Another question is if it is possible to use other forms of parallel processing, e.g. graphics processing units. This kind of hardware provides a massively parallel environment but threads are much less autonomous.

5.4 Performance Evaluation

There is no official performance benchmark for RDF engines. Previous work mainly resorts to custom made SPARQL queries or the LUBM dataset [21]. LUBM is restricted to a specific application scenario in the university domain and is not general enough in isolation. Other benchmarks in the literature are the Berlin SPARQL benchmark [16] and SP^{2}Bench [34]. We will incorporate these different benchmarks in our experiments, but will include our own set of custom queries. This way we can gear queries towards real RDF datasets and specific application patterns.

6. CONCLUSION

In this work we introduced the ternary B-tree as an alternative data structure for storing RDF. Based on the original RDF data and not on mappings thereof, we see the ternary B-tree as better prepared for the extensive string processing and ordering required by complex SPARQL queries. By decomposing triples into fixed-size elements, we regain the advantage of fast comparison. It also enables us to organize the triples into ternary search trees which gives rise to a new join strategy. As the ternary B-tree retains the natural ordering of triples, an extended partitioning scheme is possible, providing alternative ways to consider for a query optimizer and for parallelization.

7. REFERENCES

[1] Dbpedia. http://dbpedia.org.

[2] Linked data. http://linkeddata.org.

[3] Revyu. http://revyu.com.

[4] Tdb - a sparql database for jena. http://openjena.org/TDB.

[5] Resource description framework. http://www.w3.org/RDF, 2004.

[6] The 2000 u.s. census: 1 billion rdf triples. http://www.rdfabout.com/demo/census, 2007.

[7] Geonames ontology. http://www.geonames.org/ontology, 2007.

[8] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[9] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. volume 18, 2009.

[10] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ics-forth rdfsuite: Managing voluminous rdf description bases. In *SemWeb*, 2001.

[11] R. Angles and C. Gutierrez. Querying rdf data from a graph database perspective. In *ESWC*, 2005.

[12] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit"loaded: A scalable lightweight join query processor for rdf data. In *WWW*, 2010.

[13] R. Bayer and K. Unterauer. Prefix b-trees. In *ACM Transactions on Database Systems*, 1977.

[14] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *SODA*, 1997.

[15] C. Binning, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, 2009.

[16] C. Bizer and A. Schultz. The berlin sparql benchmark. 2009.

[17] M. Bröcheler, A. Pugliese, and V. Subrahmanian. Dogma: A disk-oriented graph matching algorithm for rdf databases. In *ISWC*, 2009.

[18] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: An architecture for storing and querying rdf data and schema information. In *Semantics for the WWW*, 2001.

[19] J. J. Carroll, I. Dickinson, and C. Dollin. Jena: Implementing the semantic web recommendations. In *WWW*, 2004.

[20] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient sql-based rdf querying scheme. In *VLDB*, 2005.

[21] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, 2005.

[22] S. Harris and N. Gibbins. 3store: Efficient bulk rdf storage. In *PSSS*, 2003.

[23] S. Harris and N. Shadbolt. Sparql query processing with conventional relational database systems. In *WISE*, 2005.

[24] A. Harth and S. Decker. Optimized index structures for querying rdf from the web. In *LA-WEB*, 2005.

[25] H. V. Jagadish, N. Koudas, and D. Srivastava. On effective multi-dimensional indexing for strings. 2000.

[26] D. Kolas, I. Emmons, and M. Dean. Efficient linked-list rdf indexing in parliament. In *SSWS*, 2009.

[27] T. Lampo, E. Ruckhaus, J. Sierra, M.-E. Vidal, and A. Martínez. Oneql: An ontology-based architecture to efficiently query resources on the semantic web. In *SSWS*, 2009.

[28] J. Lu, F. Cao, L. Ma, Y. Yu, and Y. Pan. An effective sparql support over relational databases. In *SWDB-ODBIS*, 2007.

[29] A. Maduko, K. Anyanwu, A. Sheth, and P. Schliekelman. Estimating the cardinality of rdf graph patterns. In *WWW*, 2007.

[30] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[31] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. In *VLDB*, 2008.

[32] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD*, 2009.

[33] E. Prud'hommeaux and A. Seaborne. Sparql query language for rdf. http://www.w3.org/TR/rdf-sparql-query, 2008.

[34] M. Schmidt, T. Hornung, M. Meier, C. Pinkel, and G. Lausen. Sp^{2}bench: A sparql performance benchmark. In *Semantic Web Information Management*, 2010.

[35] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: not all swans are white. In *VLDB*, 2008.

[36] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *WWW*, 2008.

[37] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A core of semantic knowledge unifying wordnet and wikipedia. In *WWW*, 2007.

[38] Swiss Institute of Bioinformatics. Uniprot rdf. http://dev.isb-sib.ch/projects/uniprot-rdf, 2009.

[39] J. Weaver and G. T. Williams. Scalable rdf query processing on clusters and supercomputers. In *SSWS*, 2009.

[40] C. Weiss and A. Bernstein. On-disk storage techniques for semantic web data - are b-trees always the optimal solution? In *SSWS*, 2009.

[41] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple indexing for semantic web data management. In *VLDB*, 2008.

[42] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient rdf storage and retrieval in jena2. In *SWDB*, 2003.

[43] D. Wood, P. Gearon, and T. Adams. Kowari: A platform for semantic web storage and analysis. In *WWW*, 2005.