Indexing and retrieving Semantic Web resources: the RDFStore model

Alberto Reggiori, Dirk-Willem van Gulik, Zavisa Bjelogrlic

Asemantics S.r.l., Milan, Rome - Italy, Leiden - Netherlands
{alberto, dirkx, z}@asemantics.com

Abstract

The Semantic Web is a logical evolution of the existing Web. It is based on a common conceptual data model of great generality that allows both humans and machines to work with interrelated, but disjoint, information as if it was a single global database. The design and implementation of a general, scalable, federated and flexible data storage and indexing model, which corresponds to the data model of the Semantic Web, is fundamental for the success and deployment of such a system. The generality of the RDF data model presents unique challenges to efficient storage, indexing and querying engines. This paper presents our experience and work related to RDFStore which implements a new flexible indexing and query model. The model is tailored to RDF data and is designed around the Semantic Web from the ground up. The paper describes the underlying indexing algorithm, together with comparisons to other existing RDF storage and query strategies.

Towards a lightweight database architecture

The generality of the RDF data model presents unique challenges to efficient storage, indexing and querying software. Even if the Entity-Relational (ER) data model [1] is the dominant technology for database management systems today, it has limitations in modeling RDF constructs.

RDF being unbounded, the resulting data structures are irregular, expressed using different data granularity, deeply nested or even cyclic. As a consequence, it is not possible to easily fix the "structural view" of a piece of information (object), which is instead one of the fundaments of traditional RDBMS systems trying to be much narrower and precise as possible and where an update not conforming to a single static schema is rejected. Database systems also optimize data storage and retrieval by knowing ahead of time how records are structured and interrelated and tend to use very inefficient nested SQL SELECT statements to process nested and cyclic structures.
All this is too restrictive for RDF data. Like most semi-structured formalisms [2][3] RDF is self-describing. This means that the schema information is embedded with the data, and no a priori structure can be assumed, giving a lot of flexibility to manage any data and deal with changes in the data's structure seamlessly at the application level. The only basic structure available is the RDF graph itself, which allows describing RDF vocabularies as groups of related resources and the relationships between these resources [4]. All new data can be "safely" accepted, eventually at the cost of tailoring the queries to the data. On the other side, RDF data management systems must be much more generic and polymorphic like most of dynamically-bound object-oriented systems [5]; changes to the schema are expected to be as frequent as changes to the data itself and could happen while the data is being processed or ingested.

A drawback of RDF heterogeneity is that the schema is relatively large compared to the data itself [6]; this in contrast to traditional RDBMS where the data schema is generally several orders of magnitude smaller than the data. This also implies that RDF queries over the schema information are as important as queries on the data. Another problem is that most RDF data (e.g. metadata embedded into an HTML page or RSS1.0 news feed) might exist independently of the vocabulary schemas used to mark-up the data, further complicating data structure "validation" (RDF Schema validation). This de-coupling aspect also makes the data "de-normalization" more difficult [7][8][9]. "De-normalization" is needed in RDBMS to overcome query performance penalties caused by the very general "normalized" schemas. De-normalization must be done taking into account to how the database will be used and how data is initially structured. In RDF this is not generally possible, unless all the RDF Schema definitions of the classes and properties used are known a-priori and available to the software application. Even if that might be the case, it is not a general rule and it would be too restrictive and make RDF applications extremely fragile. In the simplest and most general case, RDF software must associate the semantics to a given property exclusively using the unique string representation of its URIs. This will not stop of course more advanced and intelligent software to go a step further and retrieve, if available, the schema of the associated namespace declarations for validation, optimization or inference purposes.

It is interesting to point out that a large part of queries foreseen for Web applications are information discovery and retrieval queries (e.g. Google) that can "ignore" the data schema taxonomy. Simple browsing through the RDF data itself or searching for some sub-string into literals, or using common patterns is generally enough for a large family of RDF applications.
On the other hand, we strongly believe that RDBMS has proven to be a very effective and efficient technology to manage large quantities of well-structured data. This will continue to be true for the foreseeable future. We thus see RDF and similar less rigid, or semi-structured data technologies as complementary to traditional RDBMS systems. We expect to see RDF increasingly appear in the middle layer where lightweight systems that focus on interoperability, flexibility and a certain degree of decoupling of rigid formats are desired.
We believe that a fundamentally different storage and query architecture is required to support the efficiently and the flexibility of RDF and its query languages.

At a minimum such storage system needs to be:

- Lightweight
- Native implementation of the graph
- Fundamentally independent from data structure
- Allow for very wide ranges in value sizes; where the size distribution is not known in advance, most certainly is not Gaussian and will fluctuate wildly.
- Be efficient - it should not be necessary to retrieve very large volumes of data in order to reconstruct part of the graph.
- Allow built support for arbitrary complex regular-path-expressions on the graph to match RDF queries like RDQL [50] statement triple-patterns.
- Have some free-text support
- Context/provenance/scope or flavoring of triples

Furthermore given that RDF and the Semantic Web are relatively new, and will require significant integration and experimentation - it is important that its technology matches that of the Internet:

- Easy to interface to C, Perl and Java at the very least. Ruby, Python, Visual Basic and .NET are a pre.
- Easy to distribute (part of) the solution across physical machines or locations in order match scaling and operational habits of existing key Internet infrastructure.
- Very resistant to "missing links" and other noise.

Contexts and provenance

A RDF statement represents a fact that is asserted as true in a certain context - space - time, situation, scope, etc. The circumstances where the statement has been stated represent its "contextual" information [10][11].
For example, it may be useful to track the origin of triples added to the graph, e.g. the URI of the source where triples are defined, e.g. in an RDF/XML file, when and by whom they where added and the expiration date (if any) for the triples. Such context and provenance information can be thought of as an additional and orthogonal dimension to the other components of a triple. The concept is called in the literature "statement reification". Context and provenance are currently not included in the RDF standardisation process [48][49], but will hopefully adressed in a next release of the specification.
From the application developer point of view there is a clear need for such primitive constructs to layer different levels of semantics on top of RDF which can not be represented in the RDF triples space. Applications normally need to build meta-levels of abstraction over triples to reduce complexity and provide an incremental and scaleable access to information. For example, if a Web robot is processing and syndicating news coming from various on-line newspapers, there will be overlap. An application may decide to filter the news based not only on a timeline or some other property, but perhaps select sources providing only certain information with unique characteristics. This requires the flagging of triples as belonging to different contexts and then describing in the RDF itself the relationships between the contexts. At query time such information can then be used by the application to define a search scope to filter the results.
Another common example of the usage of provenance and contextual information is about digital signing RDF triples to provide a basic level of trust over the Semantic. In that case triples could be flagged for example with a PGP key to uniquely identify the source and its properties.
There have been several attempts [12][13][14][15] trying to formalize and use contexts and provenance information in RDF but a common agreement has not been reached yet. However, context and provenance information come out as soon as a real application is built using RDF. Some first examples are presented below.

Our approach to model contexts and provenance has been simpler and motivated by real-world RDF applications we have developed [16a][16b][16c]. We found that an additional dimension to the RDF triple can be useful or even essential. Given that the usage of full-blown RDF reification is not feasible due to its verbosity and inefficiency we developed a different modeling technique that flags or mark a given statement as belonging to a specific context. First example considers subjective assertions. The Last Minute News (LMN) [16b] and The News Blender (NB) [16c] demos allow an user rating and qualifying the source - newspapers. The user can "say" that a newspaper is "liberal" or "conservative". Of course, two users, X and Y, will show two different opinions. Without considering the context, this will result in two triples:

```
Newspaper A -> Quality -> "liberal"
Newspaper A -> Quality -> "conservative"
```

Which will be interpreted later as two quality properties assigned to the same newspaper. In reality, these triples were defined in a different contexts (which can also be expressed as triples):

```
Quality -> Defined by -> User X : Newspaper A -> Quality -> "liberal"
Quality -> Defined by -> User Y : Newspaper A -> Quality -> "conservative"
```

Next example of context needed in a Semantic Web application comes from a practical example the Image ShowCase (ISC) [16a] where RDF/XML descriptors of resources were used. RDF files were created by parsing free-formatted HTML pages. Triples were created subseq reated causing a set of wrong triples. After the correction of the RDF description (e.g. A1), new set of triples will be created, but these will be added to the old ones created form A. This will result, e.g. after a correction of the Track attribute, in something like:

```
Image X -> Track -> 3333
Image X -> Track -> 3334
```

A more correct presentation will consider the source from which the triples were derived:

```
Source RDF -> is -> A   : Image X -> Track -> 3333
Source RDF -> is -> A1  : Image X -> Track -> 3334
```

This type of context will allow removing (or ignoring) all triples created from the source RDF file A.

Last example is taken again from the Last Minute News demo system [16b]. One of newspapers use a solution which reuses URLs of articles day after day and the article - a resource - disappear after 24 hours. Triples, created for an article A, identified by URLa will be mixed with triples for article B, published a day after and identified by the same URLa. In this case, we can say, the original resource expires after 24 hours and we can decide, e.g. to make triples expire after same time. Of course, different solutions can be used here, like referencing the article "by description" instead of using URI and saving old articles but in any of these solutions we will need an information about the time where triples was defined and its expiration date. It shall be noted that this is not a specific case, in any realistic application the time component must be considered. Even if assertions about the resource can be "eternal truths", the subject - resource described may expire and disappear after some time.

A simpler example, with less theoretical implications, is simple house-keeping where knowledge about a domain can be refreshed from time to time. In this case existence of an older triple must be considered to avoid mixing of same triples defined ad different time.
Another important example of the usage of provenance and contextual information is about digital signing RDF triples to provide a basic level of trust over the Semantic Web. In that case triples could be flagged for example with a PGP key to uniquely identify the source and its properties.

Related work

Several groups have developed technology to store RDF nodes, arcs and labels into database management systems like PostgreSQL, MySQL, Oracle, IBM DB2, Interbase (and many others) and significant progress has been made. Examples are: [17][18][19][20][21][22][23][24][25][26][27][28]. Each design strikes a careful balance between flexibility, scalability, query facilities, efficiency and optimization. The major drawback of these systems is that they force RDF data into few tables having a lot of rows, resulting in multiple joins and slow retrieval time. Even though joins are relatively cheap in modern databases, the number of disk operations and query requests remains very high; even for the simplest of requests. The number of 'sub queries' needed to satisfy a single RDF query is often several orders of magnitude larger than commonly seen in RDBMS applications. Also, very often to save space DBAs design tables using significant number of indirect references, deferring to the application, or a stored procedure layer, for expanding the operation into large numbers of additional join operations just to store, retrieve or delete a single atomic statement from the database and maintaining consistency. Existing RDF implementations using object-relational database models have partially overcame this limitation by de-normalizing tables into more domain specific objects using RDF Schema taxonomy information. Such models are generally easier and more efficient to store and query, but less flexible [20][29]. The key limitation of such approach is the requirement to bundle the complete machine-readable schema definition (RDF Schema), a-priori, with the data to best structure the tables. As already explained this will rarely

be the case in the RDF world.
Due to the fact that most data is already stored into traditional RDBMS other approaches simply try to turn the problem of storing RDF efficiently upside down by producing various mapping schemas to export SQL data into RDF [30][31][32]. Other systems being more XML oriented focus on providing document centric views of RDF/XML from relational tables [33]. Recently so-called "query rewriters" started to appear; these function by directly mapping RDF graphical queries inside applications (e.g. RDQL statements) to SQL SELECT statements onto the native RDBMS and as such leaves the problem of storage efficiency, query optimization heuristics and performance to the RDBMS - leveraging existing database experience. Significant progress been made in this area by [34][35][36][37]. The SWAD-e project deliverable from Dave Beckett [38] is a good introduction.
The BerkeleyDB [39] is a good example of lightweight, portable and fast data management solution; it is suited for applications having to deal with a large amount of unbounded data of any type with an arbitrary complexity. Having no notion of schema, like traditional relation databases do, the BerkeleyDB is ideal for applications that cannot predict in advance the access and query patterns to the actual data, while needing high performance in the storage of records. The key/value paradigm also at the base of the database library results to be quite natural to store and retrieve RDF descriptions of resources having properties with some specific property values. The most common implementations use multiple B-Tree hash tables with duplicates containing pre-canned indexes of outbound nodes from a resource with a given arc, inbound nodes with a given arc and destination and the arcs between two given nodes; some implementations have additional indexes for inbound and outbound arcs of particular nodes or statement contexts [18][40][41][42]. Even if such indexes are quite fast to access and to write they only support a small subset of the possible combinations of the indexes and some queries can not be answered; such indexes use a flat storage which is in general not compact and space consuming by not avoiding repeated strings being stored. Free-text indexing of literal values is not trivial to implement unless using an off-the-shelf database or free-text indexing software.
Other approaches provide very efficient and scalable RDF storage support by using memory-mapped files on disk and ad-hoc designed in-memory data structures to hash properties and their values [43][44]. Such systems can index data quite efficiently and the size of the database can grow to the maximum address space without consuming swap space. Possible queries are limited and free-text search can only be obtained by installing third party packages.

Our solution

We have been developing a new BerkeleyDB based hashed storage which unlike most of other hashed indexers and SQL approaches uses various compressed multi-dimensional sparse matrix inverted indexes instead of B-Tree (s). Such indexes map the RDF nodes, contexts and free-text words contained into the literals to statements. There are several advantages to this approach. First, the use of a hybrid run-length and variable-length encoding to compress the indexes makes the resulting data store much more compact. Second, the use of bitmaps and Boolean operations allows matching arbitrary complicated queries with conjunction, disjunction and free-text words without using backtracking and recursion techniques. Third, this technique gives fine-grained control over the actual database content. Although our implementation of RDF indexing and storage uses custom developed software, this does not limit the potential use of the algorithm in other storage software or commercial RDBMSes. In fact our technique does not dictate a particular storage architecture beyond requiring (efficient) key/arbitrary-length-value pair operations such as GET, STORE, INCREMENT, DECREMENT and DELETE on a small number of tables.

Modeling the RDF graph data

An RDF Directed-Labeled-Graph (DLG) consists of nodes that are either resources or literals (valid Unicode strings); resources can be URI references or anonymous resources (bNodes). Nodes into a graph are connected via named resource nodes having a valid URI picked up from one or more vocabularies. A statement or triple (or fact) consists of a subject node (URI or bNode), a predicate node (URI) and an object (URI, bNode or string). Optionally, even if it is outside the RDF model, a context node (URI or bNode) might be associated to a triple.

In this section we will present a complete example taken from the RDF/XML Syntax Specification (Revised) [45] and how it can be stored and indexed into a generic hashed storage (for generic hashed storage we refer to a system supporting some algorithm for hash-addressing which provides direct retrieval of a specific row of a table or matrix. BerkleyDB systems support hashed-storage natively. Others traditional database management systems like Oracle allows to structure ad-hoc database structure as hashed-storage. Most programming languages such as Perl, Java, C++ and others also support hash-tables which could be used to store the RDF as an in-memory structure which could be stored on a secondary storage with several serialisation techniques available. Other options might be to use some XML based vocabulary to markup ad-hoc hash-table and use an XML toolkit to manage the structure) using ad-hoc designed tables to store RDF information. Particularly we will use one extra table to support free-text indexing of Unicode strings contained into RDF literal values.

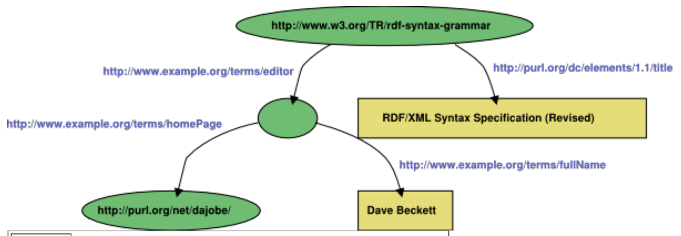
Here is the RDF/XML [45], N-Triples [46] and graphical representation of the Example 7 from the RDF M&S document [45]:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:ex="http://example.org/stuff/1.0/">
  <rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar"
                   dc:title="RDF/XML Syntax Specification (Revised)">
    <ex:editor>
      <rdf:Description ex:fullName="Dave Beckett">
        <ex:homePage rdf:resource="http://purl.org/net/dajobe/" />
      </rdf:Description>
    </ex:editor>
  </rdf:Description>
</rdf:RDF>
```

Which gives the following N-Triples:

```
<http://www.w3.org/TR/rdf-syntax-grammar> <http://purl.org/dc/elements/1.1/title> "RDF/XML Syntax Specification (Revised)" .
_:genid1 <http://example.org/stuff/1.0/fullName> "Dave Beckett" .
_:genid1 <http://example.org/stuff/1.0/homePage> <http://purl.org/net/dajobe/> .
<http://www.w3.org/TR/rdf-syntax-grammar> <http://example.org/stuff/1.0/editor> _:genid1 .
```

See the picture below for a graphical representation of the resulting RDF graph:


http://www.w3.org/TR/rdf-syntax-grammar
http://www.example.org/terms/editor
http://purl.org/dc/elements/1.1/title
RDF/XML Syntax Specification (Revised)
http://www.example.org/terms/homePage
http://www.example.org/terms/fullName
http://purl.org/net/dajobe/
Dave Beckett

st_num	
0	<http://www.w3.org/TR/rdf-syntax-grammar> <http://purl.org/dc/elements/1.1/title> "RDF/XML Syntax Specification (Revised)" .
1	_:genid1 <http://example.org/stuff/1.0/fullName> "Dave Beckett" .
2	_:genid1 <http://example.org/stuff/1.0/homePage> <http://purl.org/net/dajobe/> .
3	<http://www.w3.org/TR/rdf-syntax-grammar> <http://example.org/stuff/1.0/editor> _:genid1 .

Each statement gets a sequential number from zero on; then for any given statement number we have associated a specific bitno (offset) the gives some kind of unique key for the given component. It results to be handy to interpret such offsets in different ways in different tables. A statement context (statement group) is represented by a resource with a specific URI or bNode nodeID.

In the example above we assume that each statement has some provenance URI (statement-group, context or whatever :) which for semplicity we assume to be equal to 'http://www.w3.org/TR/rdf-syntax-grammar/example07.rdf' - then the above N-Triples maps to Quads[13] as follow:

st_num	
0	<http://www.w3.org/TR/rdf-syntax-grammar> <http://purl.org/dc/elements/1.1/title> "RDF/XML Syntax Specification (Revised)" <http://www.w3.org/TR/rdf-syntax-grammar/example07.rdf> .
1	_:genid1 <http://example.org/stuff/1.0/fullName> "Dave Beckett" <http://www.w3.org/TR/rdf-syntax-grammar/example07.rdf> .
2	_:genid1 <http://example.org/stuff/1.0/homePage> <http://purl.org/net/dajobe/> <http://www.w3.org/TR/rdf-syntax-grammar/example07.rdf> .
3	<http://www.w3.org/TR/rdf-syntax-grammar> <http://example.org/stuff/1.0/editor> _:genid1 <http://www.w3.org/TR/rdf-syntax-grammar/example07.rdf> .

For efficient storage and retrieval of statements and their components we assume there exist some hash functions which generates a unique CRC64 integer number for a given MD5 or SHA-1 cryptographic digest representation of statements and nodes of the graph as follow:

```
st0 = get_statement_hashCode('<http://www.w3.org/TR/rdf-syntax-grammar>
                               <http://purl.org/dc/elements/1.1/title>
                               "RDF/XML Syntax Specification (Revised)"
                                <http://www.w3.org/TR/rdf-syntax-grammar/example07.rdf> .' )
```

```
s0 = get_node_hashCode( 'http://www.w3.org/TR/rdf-syntax-grammar' )
p0 = get_node_hashCode( 'http://purl.org/dc/elements/1.1/title' )
o0 = get_node_hashCode( 'RDF/XML Syntax Specification (Revised)' )

st1 = get_statement_hashCode('_:genid1
                               <http://example.org/stuff/1.0/fullName>
                               "Dave Beckett"
                               <http://www.w3.org/TR/rdf-syntax-grammar/example07.rdf> .' )
s1 = get_node_hashCode( '_:genid1' )
p1 = get_node_hashCode( '<http://example.org/stuff/1.0/fullName>' )
o1 = get_node_hashCode( '"Dave Beckett"' )

st2 = get_statement_hashCode('_:genid1
                               <http://example.org/stuff/1.0/homePage>
                               <http://purl.org/net/dajobe/>
                               <http://www.w3.org/TR/rdf-syntax-grammar/example07.rdf> .' )
s2 = get_node_hashCode( '_:genid1' ) = s1
p2 = get_node_hashCode( '<http://example.org/stuff/1.0/homePage>' )
o2 = get_node_hashCode( '<http://purl.org/net/dajobe/>' )

st3 = get_statement_hashCode('<http://www.w3.org/TR/rdf-syntax-grammar>
                               <http://example.org/stuff/1.0/editor>
                               _:genid1
                               <http://www.w3.org/TR/rdf-syntax-grammar/example07.rdf> .' )
s3 = get_node_hashCode( '<http://www.w3.org/TR/rdf-syntax-grammar>' ) = s0
p3 = get_node_hashCode( '<http://example.org/stuff/1.0/editor>' )
o3 = get_node_hashCode( '_:genid1' ) = s1 = s2

c0 = get_node_hashCode( '<http://www.w3.org/TR/rdf-syntax-grammar/example07.rdf>' )
```

Here is list of the hash tables used with a little bit of explaination:

table	description	notes
MODEL	Used to store statement counters and other model misc options (free-text, source URI, compression algorithms and so on)	
STATEMENTS	Maps statements hash codes to sequential integer numbers of statements st(n)	It is generally needed to quickly check whether or not a statement is already stored in the db
NODES	Contains all the actual content more or less like in the N-Triples[46]/Quads[13] syntax	
SUBJECTS	Maps a subject resource node to a certain st(n)	Sparse matrix with compression
PREDICATES	Maps a predicate resource node to a certain st(n)	Sparse matrix with compression
OBJECTS	Maps an object node to a certain st(n)	Sparse matrix with compression
CONTEXTS	Maps resources representing contexts to st(n)	Sparse matrix with compression
LANGUAGES	Maps an object node xml:lang to a certain st(n)	Sparse matrix with compression
DATATYPES	Maps a resource node which is the rdf:datatype of an object node to a certain st(n)	Sparse matrix with compression
S_CONNECTIONS	Maps a subject resource node to statements directly or indirectly connected to it	Sparse matrix with compression
P_CONNECTIONS	Maps a predicate resource node to statements directly or indirectly connected to it	Sparse matrix with compression
O_CONNECTIONS	Maps an object node to statements directly or indirectly connected to it	Sparse matrix with compression
WINDEX	Maps Unicode case-folded representations of free-text words present into statement object literals and their stemming literals to st(n)	Sparse matrix with compression

After having parsed and ingested the above RDF/XML example statements (quads) the storage hash tables looks like as follow:

MODEL hash table		
KEY	**VALUE**	notes
counter	4	we have got 4 statements

counter_removed	0	none removed yet

STATEMENTS hash table		
KEY	**VALUE**	notes
st0	0	MODEL->{counter} is counting statements from one but we start from zero anyway
st1	1	
st2	2	
st3	3	

NODES hash table (template)										
KEY	**VALUE**									
	C data type --->	**int**	**int**	**int**	**int**	**int**	**int**	**int**	**char**	**(strings of content)**
st_num		s_len	p_len	o_len	o_lang_len	o_dt_len	c_len	st_res_len	special_byte	s p o olang odt c sr

```
where:
  s_len      = length in bytes of the subject resource identifier
  p_len      = length in bytes of the predicate resource identifier
  o_len      = length in bytes of the object resource identifier or literal string
  o_lang_len = length in bytes of the object xml:lang property value
  o_dt_len   = length in bytes of the object rdf:datatype property resource identifier
  c_len      = length in bytes of the context resource identifier
  st_res_len = length in bytes of the statement resource identifier

NOTE: possible add ons: object_xmlencoding (e.g. if different from UTF-8)

                  bits
special_byte =  01234567
                ^^^^^^^^
                ||||||||
                ||||||+--- 128 (reserved for future use)
                |||||+---- 64  (reserved for future use)
                ||||+----- 32  statement is reified
                |||+------ 16  context is bNode
                ||+------- 8   object is bnode
                ||+-------- 4   predicate is bNode
                |+--------- 2   subject is bNode
                +---------- 1   object is literal
```

Then for the above four quads we have:

NODES hash table (values for given example)	
KEY	**VALUE**
0	39, 37, 38, 0, 0, 53, 0, 1, "http://www.w3.org/TR/rdf-syntax-grammar>", "http://purl.org/dc/elements/1.1/title", "RDF/XML Syntax Specification (Revised)", "http://www.w3.org/TR/rdf-syntax-grammar/example07.rdf"
1	8, 37, 12, 0, 0, 53, 0, 3, "_:genid1", "http://example.org/stuff/1.0/fullName", "Dave Beckett", "http://www.w3.org/TR/rdf-syntax-grammar/example07.rdf"
2	8, 37, 27, 0, 0, 53, 0, 2, "_:genid1", "http://example.org/stuff/1.0/homePage", "http://purl.org/net/dajobe/", "http://www.w3.org/TR/rdf-syntax-grammar/example07.rdf"
3	39, 37, 8, 0, 0, 53, 0, 8, "http://www.w3.org/TR/rdf-syntax-grammar", "http://example.org/stuff/1.0/editor", "_:genid1", "http://www.w3.org/TR/rdf-syntax-grammar/example07.rdf"

```
NOTE: eventually nodes hash table will also store the rdf:datatype and xml:lang values (see above
      bits/bytes map explaination for nodes table)
```

Adjacency matrixes:

SUBJECTS hash table											
		VALUE									
	st_num -->	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**9**	**...**
KEY											
s0=s3		1	0	0	1	0	0	0	0	0	000
s1=s2=o3		0	1	1	0	0	0	0	0	0	000

PREDICATES hash table											
		VALUE									
	st_num -->	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**9**	**...**
KEY											
p0		1	0	0	0	0	0	0	0	0	000
p1		0	1	0	0	0	0	0	0	0	000
p2		0	0	1	0	0	0	0	0	0	000
p3		0	0	0	1	0	0	0	0	0	000

OBJECTS hash table											
		VALUE									
	st_num -->	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**9**	**...**
KEY											
o0		1	0	0	0	0	0	0	0	0	000
o1		0	1	0	0	0	0	0	0	0	000
o2		0	0	1	0	0	0	0	0	0	000
o3=s1=s2		0	0	0	1	0	0	0	0	0	000

In the simple example considered the LANGUAGES and DATATYPES hash tables are empty because the RDF triples resulting do not contain any xml:lang or rdf:datatype information - we could eventually default them to some sensible values or set them accordingly to the input RDF/XML synyax. The generated tables will store values as sparse matrixes with compression.

Until now we did not cover the statement context component, which can also be represented quite efficiently in a matrix form. For example if we assume that the above 4 statements have been stated into a context e.g. 'http://www.w3.org/TR/rdf-syntax-grammar/example07.rdf' (from which a hashcode c0 is generated), we can represent the contextual information of statements into a separated sparse matrix as follow:

CONTEXTS hash table												
		VALUE										**notes**
	st_num -->	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**9**	**...**	
KEY												
c0		1	1	1	1	0	0	0	0	0	000	all four triples are into the given context i.e. they are quads

The WINDEX hash table needs a little bit of explaination. The scope of having an additional table to index free-text words is to provide the flexibility needed to run very generic RDF queries over the database using wild-card style regular-expressions [51] on arbitrary match nodes in a graph. This is generally very useful to find a starting point where to navigate up or down the graph connections to extract meaningful data. The matrix maps the pure raw UTF-8 case-folded text representation [52] of each word present into the literal part of each statement to the correspondent statement number. The stemming of the first and last chars present in each such literals is also stored. Case-folding is used to improve query recall. This is useful in real-live applications when finding a starting point from where to navigate up or down the graph connections to extract meaningful data. Query precision is not jeopardized by this as the full, non-case folded, value is stored separately in another table. The splitting algorithm for a given object literal splits the string up into words by removing all blank spaces and special chars; each single lowercased word resulting is separately indexed for each statement. The full text of the literal is being stored separately into the NODES table above. The splitting algorithm for a given object literal splits the string up into pieces removing all blank spaces and special chars; each single lowercased word resulting is separately indexed for each statement.

The resulting WINDEX table for the above RDF/XML example is the following:

WINDEX hash table												
		VALUE										**notes**
	st_num -->	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**9**	**...**	
KEY												
dave		0	1	0	0	0	0	0	0	0	000	the word 'dave' is present in the literal of st(1)
beckett		0	1	0	0	0	0	0	0	0	000	
rdf		1	0	0	0	0	0	0	0	0	000	
xml		1	0	0	0	0	0	0	0	0	000	
syntax		1	0	0	0	0	0	0	0	0	000	
specification		1	0	0	0	0	0	0	0	0	000	
revised		1	0	0	0	0	0	0	0	0	000	
d		0	1	0	0	0	0	0	0	0	000	
da		0	1	0	0	0	0	0	0	0	000	
dav		0	1	0	0	0	0	0	0	0	000	
b		0	1	0	0	0	0	0	0	0	000	
be		0	1	0	0	0	0	0	0	0	000	
bec		0	1	0	0	0	0	0	0	0	000	
beck		0	1	0	0	0	0	0	0	0	000	
becke		0	1	0	0	0	0	0	0	0	000	stemming up to (configurable) 5 chars stemming
r		1	0	0	0	0	0	0	0	0	000	
rd		1	0	0	0	0	0	0	0	0	000	
x		1	0	0	0	0	0	0	0	0	000	
xm		1	0	0	0	0	0	0	0	0	000	
s		1	0	0	0	0	0	0	0	0	000	
sy		1	0	0	0	0	0	0	0	0	000	
syn		1	0	0	0	0	0	0	0	0	000	
synt		1	0	0	0	0	0	0	0	0	000	
synta		1	0	0	0	0	0	0	0	0	000	
sp		1	0	0	0	0	0	0	0	0	000	
spe		1	0	0	0	0	0	0	0	0	000	
spec		1	0	0	0	0	0	0	0	0	000	
speci		1	0	0	0	0	0	0	0	0	000	
re		1	0	0	0	0	0	0	0	0	000	
rev		1	0	0	0	0	0	0	0	0	000	
revi		1	0	0	0	0	0	0	0	0	000	
revis		1	0	0	0	0	0	0	0	0	000	

In the above table we do not represent the reverse stemming (last 5 chars) from the end of each indexed word for simplicity.

By using the above hash tables is possible to run very generic queries over the RDF storage but the are generally limited to match one set of statements at time. Even so, the usage of sparse matrixes to represent the indexes of RDF statements allow to run basic triple-pattern queries simply using boolean logic operations.

For example, we could run a simple query to find the resource who as title "RDF/XML Syntax Specification (Revised)" as follows:

```
find(?x,,"RDF/XML Syntax Specification (Revised)") # i.e. ?x = http://www.w3.org/TR/rdf-syntax-grammar
```

or a more generic free-text query like:

```
find(?x,, %"syn"%) # i.e. ?x = http://www.w3.org/TR/rdf-syntax-grammar
```

The result of such queries is generally stored into an iterator, which uses a sparse matrix to represent which statements match a specific query - for example the two queries above would return the following iterator structure:

st_num -->	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**9**	**...**	**notes**
result iterator	1	0	0	0	0	0	0	0	0	000	if matched st_num 0

This would allow to then scan the result set (iterator) in a second time and return the actual matching statements (or their parts) using the STATEMENTS and NODES tables. More complicated queries can be composed in a similar fashion which would match different bits of an RDF graph. The whole graph would generate a resulting iterator with bits set for any statement acutally stated into the RDF storage. Arbitrary triple-patterns can be run on RDF graphs using combinations of the above sparse matrixes and specific boolean operations.

Even if the above queries can model most of simple RDF application requirements to build simple Web pages or data conversion, it is is not generally possible to soley combine SUBJECTS, PREDICATES and OBJECTS tables to map connections between different nodes inside an RDF graph. In other words, it is not possible to search connected statements (or nodes) without using expensive and inefficient recursion algorithms. To make the RDF storage and query scaleable and guarantee that any arbitrary query can be run efficently over a stored RDF graph, a bunch of additional tables is required. Such tables must allow to map RDF statements connections and process their connections purely using boolena oprations.

This problem is generally present as soon as an application needs to aggregate and query RDF statements using a more complete query language such RDQL [50] or XPath. In that case, arbitrary nodes and statement connections need to be tracked back to the application.

For example, if we would like to run a RDQL query on the 4 statements above to get the name of the fullname of the editor of document "http://www.w3.org/TR/rdf-syntax-grammar", we would write:

```
SELECT
        ?fullname
WHERE
        ( ?document, <example:editor>,   ?editor ),
        ( ?editor,   <example:fullName>, ?fullname )
AND
        ?document eq 'http://www.w3.org/TR/rdf-syntax-grammar'
USING
        example FOR <http://example.org/stuff/1.0/>
```

By using the basic set of tables as describe above is not generally possible to get the value of '?fullName' variable without running the first query find(?document, , ?editor) and then recursively try (backtracking) on all possible values of ?editor returned on the second query find(?editor, , ?fullname) and keep track of the matching statements.

What is needed instead, is some way to efficently run the above triple-pattern searches independently and "join" then using some kind of common logical AND operation, which would return all the statement representing the biggest sub-graph matching the above query.

To overcome this problem, the RDFStore model uses 3 addional tables called S_CONNECTION, P_CONNECTIONS and O_CONNECTIONS, which sotres the connections of specific RDF nodes (resources or literals) to other nodes in the graph in an arbitrary way. In fact, a certain node in a statement (either subject, proedicate or object) can be connected to other statements via any other nodes present in the statement itself. Only resource nodes can connect to other nodes (either resources or bNodes), while literals are generally indireclty connected to other resources.

The S_CONNECTIONS hash table maps all possibile connections of a given resource node present as subject in a certain statement to all the other statements connected to the statement it belongs to via itself (e.g. if the same subject node would be the predicate or the object of another statement) or the predicate or the object (if is a resource). Here is the S_CONNECTIONS table as being stored for the above 4 statements:

S_CONNECTIONS hash table												
		VALUE										**notes**
	st_num -->	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**9**	**...**	
KEY												
s0=s3		1	0	0	1	0	0	0	0	0	000	s0 is (directly or indirectly) connected to st(0) and st(3)
s1=s2=o3		0	1	1	1	0	0	0	0	0	000	

Similarly we can generate the P_CONNECTIONS and O_CONNECTIONS table to map in the connections to the other statement components:

P_CONNECTIONS hash table											
		VALUE									
	st_num -->	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**9**	**...**
KEY											
p0		1	0	0	1	0	0	0	0	0	000
p1		0	1	1	1	0	0	0	0	0	000
p2		0	1	1	1	0	0	0	0	0	000
p3		1	1	1	1	0	0	0	0	0	000

O_CONNECTIONS hash table											
		VALUE									
	st_num -->	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**9**	**...**
KEY											
o0		1	0	0	1	0	0	0	0	0	000
o1		0	1	1	1	0	0	0	0	0	000
o2		0	1	1	1	0	0	0	0	0	000
o3=s1=s2		1	1	1	1	0	0	0	0	0	000

By using the above 3 additional tables is then possible to very efficenlty run RDQL (or XPath) queries spawming different connected statements, on the same storage, or even on different storages distributed over the Web as RDF/XML (or stored in soem kind of RDF storage). What the storage code has to do, is to simply AND logically the resulting bitmaps (bitmasks or iterators) for each triple pattern. The resulting sub-graph then would be further processed to extract the values of the single variabels as resquested by the application. The apporach can as well be used to run very efficently nested RDQL queries, disjuntion (OR) queries or exclude certain specific RDF graph branches. Coupeld with free-text indexing and contexts it turned out to be a very good compromise between RDF storage and retirval for the Semantic Web.

Even if the graph is extremely large it is generally possible to store the above (sparse) matrix efficiently by making use of some specific properties. We will briefly discuss these and the hybrid run-length/variable-length encoding algorithm used by the RDFStore software in the next paragraph.

The compression algorithm

Both the graph as well as the free-text words index are relatively sparsely populated which make simple compression possible. The bit arrays used in each can grow to very significant sizes; in the order of several, if not tens of page multiples. Combined with the intensive use of unique keys and a hashed storage a fair percentage of the data needed for any given query is unlikely to be needed often enough as to rely on natural caching of for example the disk system and the operating system. This makes compression attractive as it reduces the data volume that needs to be transferred from the relatively slow disk and/or networked storage; and instead relies on the CPU working off main memory and the CPU level cache. However as one typically operates in a stateless manner on just a few rows it is important to avoid global dictionary based compression methods. And given that the rows are relatively simple; and operated on a lot; care must be taken that any pattern search is of order(1) or better and for a "normal" row is able to (de)compress more than 1Mbyte/second on the target hardware platform. The latter is to ensure that it is still "worthwhile" versus disk-backed storage speed. Though it has been observed that even on memory based systems compression is effective. We suspect that this is due to compiler optimization with respect to the L2 or L3 cache(s) of the CPUs.
Initially a Run Length Encoding method was used; with two small optimizations. The first optimization was early termination; i.e. if the remainder of the row would solely contain zero's it would simply not list those explicitly. The second optimization was a bias towards sequences of 0's rather than 1's. With respect to the graph a number of issues make the above method not ideal.
The first issue is that certain values, such as a reference to a schema or a common property are dis-proportionally over represented; by several orders of magnitude (e.g rdf:type property or contextual information). Secondly certain other values; such as the unique reference to an item only appear once. And the final issue is that (in this release) triples and their context; i.e. the 4 values, are stored sequentially this means that not all bits are equal and certain patterns will happen more often than others.
So for this reason a variant of the Variable Run Length encoding is used along with part of the above RLE method.

This method is still applicable to the word indexing but adds the ability to recognize short patterns; and code the patterns which occur most often with short tokens, code the frequent patterns with a token which is an order to half an order shorter than the pattern and then use RLE for the remainder. At this point in time (de-)compression is such that the storage volumes are reasonable, that transfer volumes are manageable and we do not expect to give priority to work in this area. However we expect to examine this issue again and will be looking at a variant of LZ77 and/or replacement of the key pattern matching as soon as we have access to a more varied range of datasets from a wider range of operational applications.

Conclusion: RDFStore

RDFStore [47] is a perl/C toolkit to process, store, retrieve and manage RDF; it consists of a programming API, streaming RDF/XML and N-Triples parsers and a generic hashed data storage which implements the indexing algorithm as described in this paper using a few additional tables to store the raw content of the RDF nodes and other various fields (literal data types, parse type, language, statement reification and so on). RDFStore implements the RDQL query language which allows to query RDF repositories using an SQL like syntax directly from standard database interfaces like DBI, JDBC and ODBC. The storage sub-system allows transparent storage and retrieval of RDF nodes, arcs and labels, either from an in-memory hashed storage, from the local disk using BerkleyDB or from a very fast and scaleable remote storage. The latter is a fast networked TCP/IP based transactional storage library which uses multiple single key hash based BerkeleyDB files together with an optimized network routing daemon with a single thread/process per database. The API supports RDF contexts, reification, bNodes, typed and muli-lingual literals.

RDFStore has been successfully used for the development of several Semantic Web applications [16a][16b][16c] which read/write and query RDF descriptions using RDQL.

References

[1] "A Relational Model of Data for Large Shared Data Banks", E.F. Codd, Communications of the ACM, Vol. 13, No. 6, June 1970, pp. 377-387. http://www.acm.org/classics/nov95/toc.html
[2] P. Buneman, S. Davidson, G. Hillebrand and D. Suciu, "A query language and optimization techniques for unstructured data". In SIGMOD, San Diego, 1996
[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom and J. Wiener "The lorel query language for semistructured data" 1996 ftp://db.stanford.edu/pub/papers/lorel96.ps
[4] Dan Brickley, R.V. Guha "RDF Vocabulary Description Language 1.0: RDF Schema" (W3C Working Draft 23 January 2003) http://www.w3.org/TR/rdf-schema/
[5] Grady Booch "Object-Oriented Analysis and Design with Applications" p. 71-72
[6] Aimilia Magkanaraki, Sofia Alexaki, Vassilis Christophides, Dimitris Plexous "Benchmarking RDF Schemas for the Semantic Web" http://139.91.183.30:9090/RDF/publications/iswc02.PDF
[7] S. Abiteboul "Querying Semi-Structured Data" 1997 http://citeseer.nj.nec.com/abiteboul97querying.html
[8] S. Abiteboul and Victor Vianu, "Queries and Computation on the Web" 1997 http://citeseer.nj.nec.com/abiteboul97queries.html
[9] Dan Suciu, "An overview of semistructured data" http://citeseer.nj.nec.com/160105.html
[10] Graham Klyne, 13-Mar-2002 "Circumstance, provenance and partial knowledge - Limiting the scope of RDF assertions" http://www.ninebynine.org/RDFNotes/UsingContextsWithRDF.html
[11] John F. Sowa, "Knowledge Representation: Logical, Philosophical, and Computational Foundations", Brooks Cole Publishing Co., ISBN 0-534-94965-7
[12] Graham Klyne, 18 October 2000 "Contexts for RDF Information Modelling" http://public.research.mimesweeper.com/RDF/RDFContexts.html
[13] Seth Russel, 7 August 2002 "Quads" http://robustai.net/sailor/grammar/Quads.html
[14] T. Berners-Lee, Dan Connoly "Notation3" http://www.w3.org/2000/10/swap/doc/Overview.html
[15] Dave Beckett, "Contexts Thoughts" http://www.redland.opensource.ac.uk/notes/contexts.html
[16a] Asemantics S.r.l. "Image ShowCase (ISC)" http://demo.asemantics.com/biz/isc/
[16b] Asemantics S.r.l. "Last Minute News (LMN)" http://demo.asemantics.com/biz/radio/
[16c] Asemantics S.r.l. "The News Blender (NB)" http://demo.asemantics.com/biz/lmn/nb/
[17] GINF http://www-diglib.stanford.edu/diglib/ginf/
[18] Jena http://www.hpl.hp.com/semweb/
[19] Algae http://www.w3.org/1999/02/26-modules/User/Algae-HOWTO.html
[20] RDFSuite http://139.91.183.30:9090/RDF/
[21] Wraf http://wraf.org/RDF-Service/doc/html/wraf.html
[22] PARKA-DB http://www.cs.umd.edu/projects/plus/Parka/parka-db.html
[23] RDFGateway http://www.intellidimension.com/pages/site/products/rdfgateway.rsp

[24] 3Store http://sourceforge.net/projects/threestore/
[25] TAP http://tap.stanford.edu/
[26] Inkling http://swordfish.rdfweb.org/rdfquery/
[27] RubyRDF
http://www.w3.org/2001/12/rubyrdf/intro.html
[28] 4RDF http://Fourthought.com/
[29] Sesame http://sesame.aidministrator.nl/
[30] KAON REVERSE
http://kaon.semanticweb.org/alphaworld/reverse/view
[31] D2R
http://www.wiwiss.fu-berlin.de/suhl/bizer/d2rmap/D2Rmap.htm
[32] DBVIEW
http://www.w3.org/2000/10/swap/dbork/dbview.py
[33] Virtuoso http://www.openlinksw.com/virtuoso/
[34] Federate
http://www.w3.org/2003/01/21-RDF-RDB-access/
[35] Triple querying with SQL
http://www.picdiary.com/triplequerying/
[36] Squish-to-SQL
http://rdfweb.org/2002/02/java/squish2sql/intro.html
[37] Jena2 Database interface
http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/*checkout*/jena/jena2/doc/DB/index.html
[38] Mapping Semantic Web data with RDBMSes
http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report/
[39] BerkeleyDB Sleepycat http://www.sleepycat.com/
[40] rdfdb http://www.guha.com/rdfdb/
[41] Redland http://www.redland.opensource.ac.uk/
[42] rdflib http://rdflib.net/
[43] DAML DB http://www.daml.org/2001/09/damldb/
[44] ODP Search http://dmoz.org/ODPSearch/
[45] Dave Beckett "RDF/XML Syntax Specification
(Revised)" (W3C Working Draft 23 January 2003)
http://www.w3.org/TR/rdf-syntax-grammar/
[46] Jan Grant, Dave Beckett "RDF Test Cases" (W3C
Working Draft 23 January 2003)
http://www.w3.org/TR/rdf-testcases/
[47] Alberto Reggiori, Dirk-Willem van Gulik, RDFStore,
http://rdfstore.sourceforge.net
[48] Graham Klyne, Jeremy J. Carroll "Resource Description
Framework (RDF): Concepts and Abstract Syntax" (W3C
Working Draft 23 January 2003)
http://www.w3.org/TR/rdf-concepts/
[49] Patrick Hayes "RDF Semantics" (W3C Working Draft
23 January 2003) http://www.w3.org/TR/rdf-mt/
[50] Miller L., Seaborne A., Reggiori 'Implementations of
SquishQL, a simpler RDF Query Language", 1st
International Semantic Web Conference, Sardinia, 2002
[51] Mastering Algorithms with Perl By JonOrwant ,Jarkko
Hietaniemi ,JohnMacdonald 1st Edition August 1999 ISBN
1-56592-398-7 p.287
[52] Unicode Caseless Matching
http://www.unicode.org/unicode/reports/tr21/#Caseless_Matching
Last modified 2003/10/23