

Using a Generic Object Model to Build an RDFS Store

Bryan Thompson & Mike Personick
SYSTAP, LLC
{bryan, mike} @systap.com

Bradley Bebee
Science Applications International Corporation
bebeeb@saic.com

Bijan Parsia
Clark & Parsia, LLC
bparsia@isr.umd.edu

Martyn Cutcher
CTC Technology Ltd
martyn@ctc-tech.biz

Abstract – This paper presents an insert rate study for RDFS databases using secondary storage. The study explores the cost and benefits of both eager closure and truth maintenance algorithms for RDFS databases based on both OODMBS and relational technology. We review an OODBMS framework, known as the Generic Object Model, and its application to building an RDFS database and examine tradeoffs among various access path designs and their impact on an eager closure algorithm. We then measure the performance of two RDFS databases under three conditions: no closure, eager closure, and eager closure plus truth maintenance, and report on the results.

Keywords: RDF, persistence, object database, inference, truth maintenance

1. Introduction

This paper describes an OODBMS application framework called the Generic Object Model (GOM) [8,11], its application in building a fast, scalable RDFS store, and results from a scalability study undertaken to benchmark the store vis-à-vis other RDFS stores.

GOM is a flexible framework for general purpose applications requiring secondary storage and high performance. This paper reviews the GOM architecture, describes an implementation of an RDFS store as a GOM application (GOM SAIL [10]), and benchmarks the GOM SAIL and the Sesame RDBMS SAIL [13]. (SAIL is the Storage and Inference Layer defined by the Sesame framework [5].)

RDF Schema (RDFS) [1] is a semantic extension of the Resource Description Framework (RDF) [2] to allow for subclass/subproperty relations and domain/range typing on predicates. The RDF Semantics [3] document gives a model theoretic account of the semantics of RDFS, but also provides a set of entailment rules that are sound and complete for that semantics. These entailment rules are often used to guide the implementation of RDFS reasoners, especially those based on rule engines, whether forward chaining, backward chaining, or mixed. Choice of chaining strategy has both a noticeable impact on overall performance and on when the cost of reasoning is paid (eager closure pays on insert while backward chaining and magic sets can pay on query).

These strategies are present in various architectures being explored within the Semantic Web community

for building RDF and RDFS databases with secondary storage. These architectures have emerged from a variety of backgrounds including Prolog (SWIPL), datalog (Kaon-1, Kaon-2), relational (Sesame RDBMS SAIL, 3-store, and the Oracle database), and “custom” RDF store architectures (YARS, Kowari). This paper reports on an approach to building an RDFS database using the GOM. The GOM was created to support general purpose application programming and scalable orthogonal persistence, eliminating much of the impedance mismatch between object-oriented application programming and relational persistence. Although originally intended to enhance developer productivity, the GOM also exhibits excellent performance and scaling characteristics. It provides a good contrast to existing approaches and helps us to explore the suitability of various architectures.

2. Generic Object Model

The Generic Object Model (GOM) is an OODBMS application framework. It sits above a persistence layer and provides an API which is directly useful for writing applications. GOM supports data and behavior extensible persistent objects, one-to-one, many-to-one, and many-to-many associations among persistent objects, and indexed access to collection members. The basic linking mechanism is a bi-directional many-to-one association. One-to-one links are simply constrained many-to-one links. Many-to-many links are constructed using an intermediate linking object. In the UML models presented in this paper the following rules are observed: the multiplicity of the association ends is always indicated; composition is used to indicate a life cycle constraint; access from either association end is achieved using the association name; and names of association ends are ignored.

GOM is well-suited to modeling graph data. The ability to define arbitrary associations between objects in an ad-hoc fashion and have the index structures of these link sets managed automatically frees the application developer to concentrate on architecture and strategy instead of worrying about mundane tactical considerations such as the impedance mismatch between a graph model and set of relational tables.

The following shows the use of the GOM API to obtain an object manager, create an object representing an RDF graph, and to create two RDF literals and add them into the graph. It also demonstrates how to access the set of RDF literals using both unordered and indexed access mechanisms.

```
// Configure and create object manager.
IObjectManager om=ObjectManagerFactory
.newInstance( properties );

// Create some generic objects.
IGeneric graph = om.makeObject();
IGeneric lit1 = om.makeObject();
IGeneric lit2 = om.makeObject();

// Set a property on two generic objects.
lit1.set( "literal", "Hello World!" );
lit2.set( "literal", "Goodbye." );

// Collect those objects into a "literals" collection.
lit1.setLink( "literals", graph );
lit2.setLink( "literals", graph );

// Iterator will visit members of that collection.
Iterator itr = graph.getLinkSet( "literals" ).iterator();

// Iterator2 will visit members in index order.
Iterator itr2=graph.getLinkSet("literals")
.getIndex("literal").iterator();
```

2.1. Delegation patterns

The GOM encourages isolation between persistent data and business logic using a skin delegation pattern. A skin is a flyweight stateless object that implements some application interface and delegates the management of state to a persistent generic object. While multiple skins may be created for the same generic object, a one-to-one relationship is guaranteed between a skin implementing a given interface and the corresponding generic object.

The rationale for permitting multiple skins for a generic object is that applications may extend the interfaces supported by a persistent object. For example, one could define a skin to support an RSS channel and lazily create a discussion groups pertaining to any RDF value or statement in an RDF model that would serve as a nexus for collaborative work. The same RSS “skin” could be repurposed to create discussion groups for other kinds of business objects.

Another advantage that skins enjoy over subclassing is that skin implementations are independent of the GOM implementation classes. The GOM SAIL takes advantage of this feature, permitting us to run the same application code over three different implementations of the GOM architecture.

2.2. Property classes

By default the GOM is loosely typed. Any object may define any properties, any property may have any datatype, and the type of a property may change over time. The API provides three mechanisms for dealing with strongly typed data. The first is via strongly typed methods in the API, e.g., `setProperty("literal", val)` or `setProperty("inferred", true)`. The second mechanism uses property classes to describe type constraints. For example, typing a property class provides runtime type checking:

```
IPropertyClass
propertyClass=om.getPropertyClass( "literal" );
propertyClass.setTypeConstraint( String.class );
lit1.setInt( "literal", 12 );
// This throws a runtime constraint violation.
```

Finally, skins may expose strongly typed application specific interfaces for persistent data.

By default an index coerces property values to strings and then uses the default lexical ordering over those string values. If the property class has been typed, then the index will use a type-specific *Comparator* and compact datatype specific serialization of index nodes. For example the following could be used to create a strongly typed property class constraint for RDF Literals based on their datatype URI. Indices for this property class would correctly support efficient range queries, e.g., "Give me all RDF literals whose datatype URI is <http://www.w3.org/2001/XMLSchema#int> and whose value is in the range [1:10]." When necessary, a *Comparator* may be explicitly specified on the property class in order to impose an alternative collation sequence.

```
IPropertyClass
propertyClass=om.getPropertyClass(
"http://www.w3.org/2001/XMLSchema#int" );
propertyClass.setTypeConstraint( Integer.class );
```

Association classes may be used to specify multiplicity constraints, constraints on the supplier and/or client of the association, and life cycle (composition) constraints using properties similar to those in the Unified Modeling Language (UML) [4].

2.3. Object cache

The GOM guarantees that there is a one-to-one relationship between live objects in a given execution context and the corresponding persistent objects. This guarantee is realized by a weak reference cache for "live" objects. Each operation directed to persistence layer by the application touches the cache.

The cache is structured as an internal hard reference LRU cache policy of capacity N and an outer weak reference cache. Operations touching the outer cache also touch the inner cache, causing the LRU ordering to be updated, but do not write through to secondary storage. Objects are buffered in memory until (a) they are no longer referenced by the application; and (b) they are evicted from the LRU cache. While keys (object identifiers) are removed from the weak cache once the garbage collector has cleared the weak reference, eviction notices are generated by the inner cache. In practice, this means newly created objects, strongly reachable objects and recently used objects do not require any I/O. In our experience, GOM applications may be CPU bound since access patterns result in a significant percentage of relevant persistent objects living in cache. This makes it possible to write code, such as the reasoner, which touches a large number of objects without becoming I/O bound.

The guidance for the Java garbage collector is that weak references should be cleared as if no reference exists while soft references should be maintained until the garbage collector determines that it needs more memory (the policy is fairly flexible). This means that soft references cause objects to be retained longer. Based on observation, the use of a soft reference outer cache policy leads to unnecessary retention of objects and longer commit processing since fewer objects are incrementally written to stable storage before the transaction commits.

3. RDFS application

The GOM SAIL uses the GOM for persistent object management and Sesame's generic architecture for storing and querying RDF [5]. As such, the GOM SAIL may be used as a plug and play alternative for the back ends bundled with the Sesame 1.x distribution. The GOM SAIL has three main components: an implementation of the Graph API (manipulation of RDF graphs), an implementation of the Storage And Inference Layer (SAIL), and an RDFS reasoner used to compute and maintain the closure of the store according to the model-theoretic semantics specified by RDF Model Theory [3]. The strategy used for inference and truth maintenance is similar to the strategy described in "Inferencing and Truth Maintenance in RDF Schema" [6].

The data model for RDF Values and Statements and their relationship to the RDF Graph are straightforward [Figure 1]. Each statement has a subject, predicate, and object property. These properties are constrained to be many-to-one associations (many

Statements may have the same Value for their subject, predicate or object role, but each Statement has only one value in each of those roles). A Statement has a type property which identifies whether it is an axiom, inferred by the reasoner, or explicitly given to the store. Finally a Statement has a key property which is used in the primary statement index. This key is formed by a concatenation of the Object Identifier (OID) of the subject, predicate and object role for that statement.

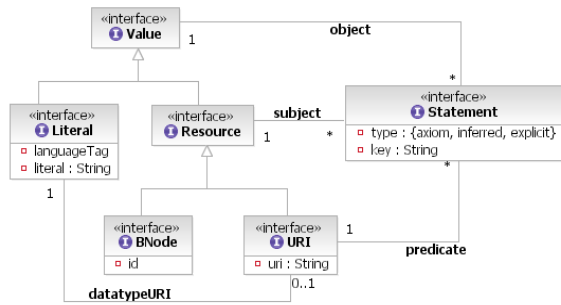


Figure 1 RDF Statement model

The concrete implementations of the interfaces in Figure 1 are flyweight stateless skins that implement the OpenRDF Graph API but delegate management of state to underlying persistent generic objects. Statements are skinned as such, and RDF Values are skinned as either Literals, BNodes, or URIs according to what properties are bound to the particular underlying generic object. Per the delegation pattern described in Section 2.1, skins and their persistent objects maintain a one-to-one relationship, enforced by a skin factory using a WeakReference cache.

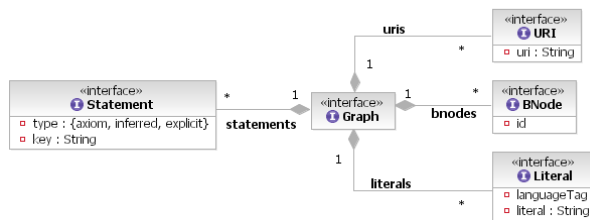


Figure 2 Collections maintained by the Graph

The Graph maintains several collections which are modeled as link sets [Figure 2]. These link sets correspond directly to the URIs, BNodes, Literals, and Statements in use by the Graph. The association classes for these collections have the composition attribute set, ensuring that the life time of the objects in these collections is bounded by that of the Graph.

RDF uses distinct lexicons for URIs, BNodes and Literals. These lexicons are modeled by the uris, bnodes, and literals collections respectively. Lookup for an RDF Value of a known type is straight-forward.

URI	graph.getLinkSet("uris").getPoint(val);
BNod e	graph.getLinkSet("bnodes").getPoint(val);
Literal	graph.getLinkSet("literals").getPoint(val);

In each case we access the appropriate link set for the Value type and use the index for the appropriate property to locate the pre-existing RDF Value. (Lookup of literal values is slightly complicated by RDF's distinction between literals with a language tag and literals with a datatype URI.)

The following code fragment demonstrates how one would create a statement using a new subject, predicate and object and then insert it into a graph. In practice this operation is encapsulated by the flyweight skins that implement the Graph API and a ValueFactory that handles uniqueness constraints on RDF Values and Statements.

```

IGeneric graph = ...;

IGeneric subj = graph.makeObject();
subj.setString("uri", "http://.../john");
subj.setLink("uris", graph);

IGeneric pred = graph.makeObject();
pred.setString("uri", "http://.../love");
pred.setLink("uris", graph);

IGeneric obj = graph.makeObject();
obj.setString("uri", "http://.../mary");
obj.setLink("uris", graph);

IGeneric stmt = graph.makeObject();
stmt.setString("key",
subj.identity()+"|" +pred.identity()+"|" +obj.identity());
;
stmt.setInt("type", EXPLICIT);
stmt.setLink("subject", subj);
stmt.setLink("predicate", pred);
stmt.setLink("object", obj);
stmt.setLink("statements", graph);

```

3. 1 Truth Maintenance

To support efficient truth maintenance on statement deletion under the closure strategy employed by the store and its reasoner, three new association classes were added to model the justification chains linking statements, both explicit and inferred, to the statements that license them. The entailment rules described in RDF Model Theory [3] require either one or two antecedents, modeled as *support1* and *support2*, and

have a single consequence, coded by the *conclusion*. During forward chaining, when an entailment rule is triggered by the presence in the knowledge base of its required supports, a Proof object is created and associated with those supports and with the conclusion they license. Conclusions may obviously support further inferences, thus creating chains of justifications. Special care must be taken to eliminate any cycles in justification chains. When an explicit Statement is deleted from the Graph but is supported by at least one Proof, its type is simply changed from explicit to inferred. If the Statement is no longer supported by any Proofs, then it is deleted. If it is deleted, then its justification chains are traversed and all inferences licensed by it are similarly examined and deleted if they are no longer supported.



Figure 3 Truth maintenance

3.2. Access patterns

The basis of RDF query is the ability to rapidly identify a Statement, or a set of Statements, based on a pattern in which zero or more values may be unbound. For example, “Who does John love?” would be posed as (John:love:?). A system must provide an efficient access path for each the following access patterns:

spo	?po	s?o	sp?	??o	S??	?p?	???
-----	-----	-----	-----	-----	-----	-----	-----

Simple traversal of the link set for the statements association on the Graph provides optimized query answering for the (?:?:?) access pattern. The (s:p:o) and (s:p:?) access patterns are answered by point and range queries, respectively, on the primary SPO index:

?:?:?	graph.getLinkSet("statements").iterator();
s:p:o	graph.getLinkSet("statements").getIndex("key").getPoint(s + " " + p + " " + o);
s:p:?	graph.getLinkSet("statements").getIndex("key").getRange(s+" " +p+" " ,s+" " +p+"").iterator();

When a single role in the access pattern is bound, the appropriate association from the RDF Value provides an optimized access path:

s:?:?	s.getLinkSet("subject").iterator();
?:p:?	p.getLinkSet("predicate").iterator();
?:?:o	o.getLinkSet("object").iterator();

The final two access patterns, (s:?:o) and (?:p:o), required some exploration to determine an efficient access strategy. The costs and benefits were explored

for three different approaches: traditional B+-tree indices (declaring SOP and POS indices to service these access patterns), link set joins (computing the intersection of link sets – see below), and PO and SO 2-tuple lexicons (PO and SO collections containing exactly those Statements required to answer any given (?:p:o) and (s:?:o) access pattern query). Link set joins proved to be the most effective means of answering these access patterns:

s:?:o	s.getLinkSet("subject").getJoin().add(o.getLinkSet("object")).getIntersection();
?:p:o	p.getLinkSet("predicate").getJoin().add(o.getLinkSet("object")).getIntersection();

A link set join provides very efficient filtering for n-dimensional link set intersection. The algorithm is straightforward and relies on two characteristics of the GOM: (1) finding the size of a given link set is O(1); and (2) given a generic object and a link set, the cost of a link set membership test is O(1). Given these characteristics, the link set with the smallest size is selected – this represents the minimum necessary variety. The members of that link set are then scanned, filtering out members that are not also members of the N-1 other link sets. It was not clear a priori whether link set joins would be an effective technique for the RDF access patterns (?:p:o) and (s:?:o). In practice, it seems the minimum variety is often small for high-dimensional data, making these joins highly effective. The results in this study use link set joins supplemented by an SPO index, but this strategy performs only marginally better than using link set joins for all access paths.

4. Performance study

The performance of the RDFS prototype (using CTC GOM release date 12/9/2005 [7,8]) was compared to the Sesame 1.2.1 RDBMS SAIL (using MySQL 5.0.16). Both systems use secondary storage and have the option of computing the eager closure of explicit triples loaded into the database and of using a truth maintenance algorithm to maintain that closure under statement removal. The SAIL integration provides common features sets, especially data loading and RDQL query, and common overhead for the access path test. The JVM was BEA JRockit 1.4.2_05 with “-server” (the default VM for JRockit) and “-Xms2048m” to grant the JVM process 2G of RAM.

The tests were performed on a Dell PowerEdge 1850 (service tag 21P3481). The machine has four 3.2Ghz Xeon processors and 8GB RAM. Programs and the operating system were on the built-in 146GB SCSI disks, but data were on an external 800G RAID 1

SCSI drive array using an on-board RAID controller. The operating system was Red Hat Enterprise Linux 3 (Kernel 2.4.21-4.ELsmp).

The RDBMS client and server communicated using a localhost JDBC connection. While a relational database creates a client – sever division and runs in two processes, no system tested was observed to take advantage of the SMP feature of the server, and the server load during tests did not exceed 25% CPU utilization across all CPUs and would have similar performance on a non-SMP machine.

4.1. Insert rate test

The insert rate test focused on the costs and benefits of eager closure as revealed by the GOM and RDBMS SAILS. For this test, we loaded the same data set under three different conditions for each store: 1) eager closure and truth maintenance; 2) eager closure without truth maintenance – proofs are not stored; and 3) RDF-only semantics – inferences are not computed. The first two conditions are interesting since truth maintenance is required only for efficient update on statement removal. A Semantic Web application with append-only semantics does not benefit from truth maintenance. The third case provides a basis for measuring the potential improvement that could be realized by a store using query time inference to provide RDFS semantics and helps us characterize the costs associated with eager closure.

Table 1 Insert rates

Store	SAIL	TM	Load time	Stable time	Store size	Triples / sec.	MB / sec.
GOM	RDFS	yes	256	257	347	1,883	1.35
GOM	RDFS	no	180	181	115	2,674	0.64
GOM	RDF	n/a	97	97	75	2,821	0.77
MySQL	RDFS	yes	154	318	199	1,522	0.63
MySQL	RDFS	no	155	249	87	1,944	0.35
MySQL	RDF	n/a	152	166	46	1,648	0.28

The data set for this experiment was Wordnet [9]. The knowledge base consisted of 273,644 explicit triples in two files (one schema and one data) and resulted in 210,225 inferred triples – an expansion of 77%. When eager closure was enabled, the GOM store computed 1,362,925 Proof objects.

TM indicates whether or not truth maintenance was supported and hence whether or not proofs were stored on disk. *Load time* is the time spent reading the source data set as reported by the Sesame administration

interface. *Stable time* is the time required until the data are stable on disk as reported by the end of the SAIL transaction. *Store size* is the size of the persistent data on disk in megabytes. *Triples/sec* is the number of triples ingested per second (including inferences). This value was computed using the actual number of triples written to the store (including axioms and inferences where appropriate) divided by the Stable time. *MB/sec* was computed based on the store size and the stable time.

Notice that the GOM store has nearly no latency between the Load time and the Stable time. This reflects the incremental commit policy of the GOM store and its interaction with the weak reference object cache as described Section 2.3. I/O operations are interleaved with application processing, resulting in an extremely short latency for the actual commit. In contrast the RDBMS SAIL performs significant processing once the SAIL has finished “loading” triples.

The cost of the “commit” for the RDBMS SAIL is broken down further in Table 2. First, the raw commit processing time of the MySQL database can be observed from the last condition in Table 1 since no inference was performed. This is nearly 9% of the total time required to load the data set. Second, the eager closure algorithm for the RDBMS SAIL adds 50% to the time of loading only the explicit triples, but expands the knowledge base by 77% (in number of statements), meaning it is actually cheaper to infer a triple than it is to read it from a data set. Finally, persisting the information necessary for the truth maintenance algorithm for the RDBMS SAIL adds another 28% over the time required to compute the eager closure and doubles the storage requirements.

The GOM store has different characteristics. The commit is nearly instantaneous. Computing the eager closure adds 87% to the time required to load the raw triples – this is very close to the expansion of the knowledge base in number of statements under inference. Finally, storing the proof objects adds an additional 42% to the time required to compute the eager closure, but triples the storage requirements. Note that the proofs are always computed if the eager closure is computed. The difference in this condition is whether or not the proofs are persisted.

Table 2 Breakdown of costs

Feature	GOM		MYSQL	
	time	space	time	space
Truth maintenance	42%	202%	28%	129%
Eager closure	87%	53%	50%	89%
Commit	0%		9%	

Based on Table 2, we can infer that statements are stored more compactly in the GOM store, but that the MySQL store provides a more compact representation of the proofs. This suggests that a significant performance gain could be realized in the GOM store by a more compact representation of the proofs, which will be the subject of future work. The impact of the truth maintenance algorithm can also be seen in Table 1 since the average I/O write rates are significantly higher for the GOM store. This runs counter to common wisdom since MySQL is known as a fast RDBMS platform while the GOM store is 100% Java. While the RDBMS can rely on pre-declared schemas and fixed size records to optimize storage for statements proofs, the GOM uses a generic data record. A generic object may have any number of properties and new properties may be defined at any time. GOM achieves compact storage using a highly tuned and extensible serialization framework [12].

The cost of computing the eager closure for both stores appears to be reasonable given that it reduces the complexity of query answering to query against ground terms in the database. Persistence of justifications for truth maintenance adds a significant cost, and applications with append-only semantics may choose to not use truth maintenance in exchange for a faster and smaller database.

6. Conclusions and future work

The Generic Object Model is an easy-to-use application framework for building high performance and scalable Semantic Web applications. The GOM-based solution compares favorably to mature relational database implementations, both in terms of performance and flexibility. The technique of eager closure accepts some additional cost when loading RDFS data but reduces the complexity of RDFS query answering to that of RDF – simple queries against the database. When using an eager closure strategy, truth maintenance under deletion for RDFS is a meaningful approach for applications that will perform frequent updates and deletions on the repository, but it is an expensive option that can be disabled for applications which expect append-only semantics for their store.

The RDFS prototype is open source [10,11]. This paper reports on the performance of the CTC GOM store [7,8]. There is also an open source stack for the persistence layer [11] with similar performance characteristics, but currently without support for very long running transactions or full transactional isolation.

We are interested in exploring the intersection of architecture, expressivity, and scale. We are currently exploring several questions related to scalable Semantic Web applications, including a comparison of eager and query time inference strategies, query optimization for RDFS, and high concurrency designs for RDFS databases. We hope to benchmark additional RDFS databases, especially those based on non-relational technologies. In related work we are exploring a high-performance computing application of the GOM to common sense reasoning.

7. References

- [1] Brickley, D., and Guha, R.V., *RDF Vocabulary Description Language 1.0: RDF Schema*, World Wide Web Consortium, 2004.
- [2] Klyne, G., and Carroll, J.K., *Resource Description Framework (RDF): Concepts and Abstract Syntax*, World Wide Web Consortium, 2004.
- [3] Hayes, P., *RDF Semantics*, World Wide Web Consortium, 2004.
- [4] *Unified Modeling Language*, Object Management Group, Inc., 2006.
- [5] Broekstra, J., Kampman, A., and Harmelen, F., “Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema,” *International Semantic Web Conference*, Sardinia, Italy, 2002.
- [6] Broekstra, J., Kampman, A., “Inferencing and Truth Maintenance in RDF Schema: Exploring a Naive Practical Approach,” *Second International Semantic Web Conference*, Sanibel Island, Florida, USA, 2003.
- [7] Thompson, B., *GOM for CTC*, CognitiveWeb, 2006. See <http://proto.cognitiveweb.org/projects/cweb/multiproject/cweb-generic-ctc/index.html>.
- [8] The Generic Object Model is part of a suite of Java-based technology created by CTC Technology Ltd, a UK company. See <http://www.cutthecrap.biz/software/whitepapers/theproject.html>.
- [9] *WordNet 2.1*, Princeton University, 2005.
- [10] Personick, M. and Thompson, B., *GOM SAIL*, CognitiveWeb, 2006. See <http://proto.cognitiveweb.org/projects/cweb/multiproject/cweb-rdf-generic/index.html>.
- [11] Thompson, B., *GOM (Native)*, CognitiveWeb, 2006. See <http://proto.cognitiveweb.org/projects/cweb/multiproject/cweb-generic-native/index.html>.
- [12] Thompson, B. *Extensible Serialization Framework*, CognitiveWeb, 2006. See http://sourceforge.net/project/showfiles.php?group_id=95673&package_id=188700.
- [13] Broekstra, J., Kampman, A., *Sesame*, 2006. See <http://www.openrdf.org>.