Owlgres: A Scalable OWL Reasoner

Markus Stocker and Michael Smith

Clark & Parsia LLC, Washington, DC, USA
{mstocker,msmith}@clarkparsia.com

Abstract. We present Owlgres, a DL-Lite reasoner implementation written for PostgreSQL, a mature open source database. Owlgres is an OWL reasoner that provides consistency checking and conjunctive query services, supports DL-Lite$_R$ as well as the OWL *sameAs* construct, and is not limited to PostgreSQL. We discuss the implementation with special focus on *sameAs* and the supported subset of the SPARQL language. Emphasis is given to the implemented optimization techniques which resulted in significant performance improvement. Based on a confidential NASA dataset and part of the DBpedia dataset, we show a typical use case for Owlgres, i.e. given a terminology and a dataset, Owlgres provides querying on a persistent knowledge base with reasoning at query time in the expressivity of DL-Lite$_R$.

1 Introduction

The thoughts of Tim Berners-Lee about Linked Data Design Issues [3] have recently found traction in the Linked (Open) Data[1] (LOD) community, which aims to connect related data using Web and Semantic Web principles and technologies. As shown in the Linking Open Data dataset cloud[2], the number of linked datasets is growing. Together with the growing number of linked datasets, the number of triples available on the Semantic Web is growing too. Events like the Semantic Web Billion Triples Track[3] colocated with the International Semantic Web Conference 2008 (ISWC2008) or the LOD Triplification Challenge[4] colocated with the I-SEMANTICS'08 conference[5] further underlie the current trend within the Semantic Web community of exposing RDF [14] data on the Web.

The availability of RDF data on the Web is certainly a fundamental goal, but it is only a first step. An important and still open question is whether something meaningful, e.g. efficient querying, can be performed over the data.

Although, currently the focus within the Semantic Web community is on *getting data out*[6], aiming at the re-use and re-combination of data, the Semantic

[1] http://linkeddata.org/
[2] http://richard.cyganiak.de/2007/10/lod/
[3] http://iswc2008.semanticweb.org/calls/call-for-semantic-web-challenge-and-billion-triples-tracks/
[4] http://triplify.org/Challenge
[5] http://triple-i.tugraz.at/i_semantics
[6] http://www.youtube.com/watch?v=6eGcsGPgUTw

Web includes a powerful layer on top of RDF which enables automated reasoning. Hence, in addition to data integration, automatic inference of implicit knowledge may be a useful tool for the Semantic Web.

Owlgres aims at both efficient querying over a scalable persistent store and automatic reasoning for RDF and OWL data.

The paper is organized as follows. In Section 2, we succinctly review the DL-Lite family and discuss some implementation details with special emphasis on optimization techniques. In Section 3, we discuss two use cases for Owlgres which show a typical usage of efficient querying and reasoning over persistent RDF data. In Section 4, we briefly discuss related work. Finally, in Section 5, we draw some conclusions and talk about future directions.

2 Implementation: Some Details

In this section, we discuss some details about the Owlgres implementation. Specifically, in Section 2.1, we briefly review DL-Lite and reference to the relevant publications. In Section 2.2, we discuss the optimizations implemented in Owlgres that have proved to be useful in minimizing the number of queries returned by the DL-Lite reformulation. Finally, in Section 2.3, we highlight our implementation for OWL *sameAs* and annotations [2].

2.1 The DL-Lite Family

DL-Lite [6] is a family of description logics (DLs) tailored for tractable reasoning and efficient query answering. The different logics of the family are fragments of expressive DLs and show some interesting properties. First, they are rich enough to capture the expressivity required for a number of specialized ontology languages (e.g. UML). Second, in addition to supporting standard reasoning services (e.g. subsumption), they are designed for conjunctive query answering over an ABox maintained in secondary storage (typically a RDBMS). Third, they show interesting computational properties, i.e. the standard reasoning tasks are polynomial in the size of the TBox[7] and query answering is LOGSPACE in data size.

For a discussion on the syntax and semantics of DL-Lite DLs we refer to [6] as a detailed review of the languages is not within the scope of this paper. For the purpose of this paper, however, we briefly highlight the query reformulation process used for DL-Lite$_{\mathrm{R}}$ query answering.

Query answering in DL-Lite$_{\mathrm{R}}$ mainly consists of two steps. In the first step, the conjunctive query provided by the user is reformulated based on the TBox axioms. Hence, the user's conjunctive query is expanded such that it encodes the relevant TBox knowledge. (Note that the query is expanded to a collection of queries and not in the number of query atoms.) For instance, assume a terminology with two concepts C and D and an axiom which states that C is a

[7] For comparison, OWL-DL, i.e. the $\mathcal{SHOIN}$ DL, is NExpTime-complete for concept satisfiability and ABox consistency

subclass of D, i.e. $C \sqsubseteq D$. If the user's conjunctive query is $\{D(?x)\}$, i.e. a set consisting of a query atom that asks for the individuals of the concept D, the reformulation algorithm encodes the knowledge about C being a subclass of D into the query. This is what is meant by reasoning at query time. We know by the terminology that instances of C are also instances of D. Hence, we expand the conjunctive user query $\{D(?x)\}$ with a conjunctive query consisting of the query atom $C(?x)$.

In the second step of DL-Lite$_\mathrm{R}$ query answering, we discard the TBox and the reformulated conjunctive query is evaluated over the ABox. In our working example, the expanded conjunctive query $\{\{D(?x)\}, \{C(?x)\}\}$ explicitly queries for all the individuals of C or D. Hence, the overall result of the user query is the union query of the reformulated sets of query atoms.

In Owlgres, the second step consists of three additional steps. First, the reformulated conjunctive query is optimized (see Section 2.2) with the goal of minimizing the set of queries that eventually hit the database. Second, the optimized reformulated conjunctive query is translated into SQL such that the request can be evaluated by PostgreSQL. Finally, in the third step, we process the result set returned by PostgreSQL and possibly apply further query constraints that have not been written into the SQL query.

2.2 Optimizations

Owlgres currently implements three types of optimizations: query simplification, selectivity optimization, and the formulation of a set of conjunctive queries as a single SQL UNION query.

Query simplification is performed during the reformulation process and is applied to the initial conjunctive query, as well as every expanded conjunctive query returned during the reformulation process. The goal of query simplification is to simplify a conjunctive query by removing atomic query atoms, e.g. $C(?x)$, that are redundant because of either role domains or role ranges. More precisely, given a conjunctive query q the domain simplifier removes all atomic query atoms from q that are redundant in q because they define a constraint which is implicitly defined by the domain of a query *role* atom. For instance, if the conjunctive query q is $\{\{C(?x)\}, \{R(?x, ?y)\}\}$ and the domain of the role R is known to be of type C, then the domain simplifier drops the atomic query atom $C(?x)$ from q as the information that the individuals bound to the variable $?x$ are instances of C is redundant. By removing such atomic query atoms, we reduce the number of atoms in some queries, which may significantly affect the number of reformulated conjunctive queries.

The second optimization is based on the concept of selectivity. Selectivity of a condition is defined by Piatetsky-Shapiro and Connell in [15] as the "fraction of tuples that satisfy the condition". In Owlgres, we are particularly interested in knowing which query atoms or conjunctive queries have zero selectivity. If a query atom in a conjunctive query has zero selectivity, then the entire conjunctive query has zero selectivity, i.e. no tuple in the database will match the conjunctive query. Hence, conjunctive queries with zero selectivity can be safely removed from the

reformulated conjunctive user query. This requires knowing which concepts and which roles in the terminology have zero selectivity.

There are two cases where selectivity optimization is applied in Owlgres. [7] mentions a rewriting technique for qualified existential quantifications (i.e. $\exists R.C$) where auxiliary roles are introduced. In Owlgres, we use this rewriting technique to deal with qualified existential quantifications. Auxiliary roles are generated by Owlgres internally during TBox loading for each encountered qualified existential quantification. Naturally, auxiliary roles will not be involved in any ABox assertion and, hence, have zero selectivity. This is important as the reformulation may introduce auxiliary roles in the expansion of a conjunctive query, but conjunctive queries with at least one auxiliary role have zero selectivity and, hence, can be discarded.

This first case of selectivity optimization can be generalized to any concept or role name defined in the terminology. While in the first case of auxiliary roles the zero selectivity is known without any computation, in the second case we need to mantain counters for the frequency of TBox terms used in the ABox. This is done during ABox loading which, in Owlgres, is a separate process from (and performed after) TBox loading. Practically speaking, only terms that are known to have zero selectivity are relevant for the optimization. As for auxiliary roles, we perform this selectivity-based optimization after the reformulation process to minimize the set of reformulated queries. The goal is to identify conjunctive queries with at least one term (concept or role) that has a known zero selectivity, in which case we can safely drop the corresponding conjunctive query from the reformulated set of conjunctive queries.

Finally, the optimized reformulated query is translated into a single SQL UNION query. This has at least two advantages. First, a single SQL query avoids multiple connections to the database, i.e. JDBC overhead. Second, we exploit the native optimizer by providing a complex query that may be optimized by the RDBMS.

Statistics about the frequency of terms do not come for free. We need to maintain them during updates and deletes. Unfortunately, the native PostgreSQL optimizer seems to be unable to identify zero selectivity conjunctive queries, which is probably caused by the fact that RDBMS typically use probabilities for selectivity estimation and, hence, lack exact information. We speculated about extending the native PostgreSQL optimizer such that it would have access to exact figures. However, it was hard to estimate the cost of this effort and the extension would tie Owlgres to PostgreSQL, at least with respect to selectivity optimization.

To give an idea about the effect of the implemented optimizations described above, we evaluated the techniques on a concrete dataset and a set of queries. We use the Lehigh University Benchmark [10] (LUBM) and its associated SPARQL [16] queries. (Note that LUBM, as generally used, is not expressible in DL-Lite because it uses transitive properties and because qualified existentials are used in the left side of subsumption relations. The results presented here are with a modified version that reduces the expressivity.) Figure 1 shows the performance

of the 14 LUBM queries on the *University0* dataset (i.e. roughly 100k triples). The timings are in milliseconds on a logarithmic scale. We first show the timing for the non-optimized reformulated set and then the timing for the optimized set.

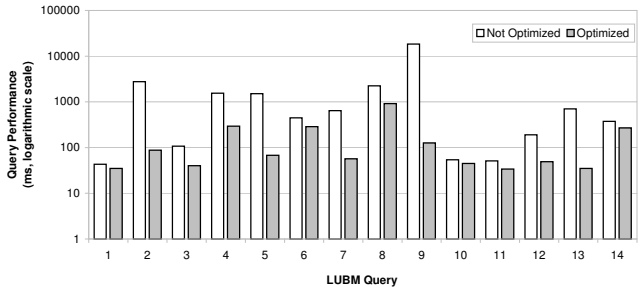

Fig. 1. Effect of the optimizations on LUBM

For instance, if we apply the optimizations described above to the LUBM query 9 we reduce the initial reformulated set of 726 queries to just two queries. These two queries are responsible for the final result set. By creating an SQL UNION of the two queries we eventually execute a single SQL query. The LUBM query 9 takes 18 seconds to be executed *without* the optimizations and 0.1 seconds *with* the optimizations. Identifying auxiliary roles is the main source of optimization in this case.

The following short example highlights a case where selectivity optimization does an important job. The case is based on the DBpedia[8] Infobox data and a query for book titles. The DBpedia TBox is based on the YAGO [17] terminology which is, in turn, based on WordNet [9]. Like WordNet, YAGO is deeply ramified. Because the individuals defined by DBpedia are for the vast majority instances of leaf concepts of the YAGO taxonomy, there are many YAGO terms that have zero selectivity, even on a large dataset like the DBpedia Infobox data. To make the point more clear, we provide some figures based on the query (here abbreviated for the sake of readability) $\{\{Book(?s)\}, \{name(?s, ?n)\}\}$ which asks for the titles of books. We query over the DBpedia Infobox dataset, i.e. roughly 24 million triples (2,198,649 concept assertions and 21,503,343 data role assertions). The TBox contains totally 58,108 concepts and 53,898 (i.e. around 93%) of them have no asserted instances, i.e. zero selectivity. Without the selectivity optimization, the reformulated set size for the query above is 83, i.e. the overall result set is computed as a UNION query of 83 conjunctive queries. With the selectivity optimization, the reformulated set size for the query is 16, i.e. only 16 queries out of 83 are responsible for the final result set. There is, hence, no need to stress the database with 67 superflous queries. Without the optimizer, the query runs in 61,743 ms and with the optimizer in 14,336 ms.

[8] http://dbpedia.org/

2.3 Support for OWL sameAs and Annotations

Owlgres supports the OWL *sameAs* construct by preprocessing. In a nutshell, OWL *sameAs* assertions are processed during ABox loading such that Owlgres has correct information to handle the queries without doing any OWL *sameAs* reasoning at query time. OWL *sameAs* support requires an extension to the basic relational schemas since we need to store more information. In particular, for ABox assertions we store both the asserted individual as well as the canonical individual, where the canonical individual corresponds to the individual returned by the computation of the OWL *sameAs* (transitive) closure for the asserted individual after normalizing *sameAs*, i.e. after enforcing an ordering on the individuals involved in the closure. For instance, assume an ABox with the assertions $C(a)$, $C(b)$, $C(c)$, $R(a, r)$ and $R(c, s)$. Moreover, it is asserted that the individual a is *sameAs* b and the individual b is *sameAs* c. Depending on the order in which the ABox assertions are processed, a final state for the three concept assertions could be the one listed in Table 1.

Asserted	Canonical
$C(a)$	$C(a)$
$C(b)$	$C(a)$
$C(c)$	$C(a)$

Table 1. Canonical representation of OWL *sameAs*

Because of the transitive OWL *sameAs* relation between the individuals a, b and c, the canonical individual for each of them is, in our example, the individual a.

Naturally, we have to consider the extended schema for OWL *sameAs* in query translation to SQL. In particular, relational joins that are required for conjunctive queries are performed over the canonical rather than the asserted individual. For instance, for the conjunctive query $\{\{C(c)\}, \{R(c, ?x)\}\}$ we have to join over the canonical individual for c such that the result set contains both $?x = r$ and $?x = s$.

Note that support for OWL *sameAs* does not come for free. We have to maintain the closure during updates and deletes which is one reason why we store both the asserted and canonical individual. Moreover, with OWL *sameAs*, reasoning is performed in two modes: at load time for the OWL *sameAs* closure and at query time for the DL-Lite query reformulation.

Owlgres also supports OWL annotation properties. Although annotation properties have no real semantics in OWL DL, they are important and often used in queries. A simple example is *rdfs:label* and a query that asks for the resource with a given label. In order to meet this important requirement, we decided to support OWL annotations in Owlgres from the beginning. As required by the specification [2], we handle OWL annotation properties strictly separated from OWL object and datatype properties. Furthermore, because of the differ-

ent nature of OWL annotation properties with a data literal and URI reference object, Owlgres manages them in separate relations.

2.4 SPARQL Support

Owlgres supports SPARQL [16] as a language to formulate queries over knowledge bases. In Owlgres, the fundamental part of SPARQL are Basic Graph Patterns (BGP). A BGP is translated into a conjunction of query atoms. For instance, the (abbreviated) BGP { `?x rdf:type ub:Student . ?x ub:name ?n` } is translated into the conjunctive user query with two query atoms, i.e. {{ `Student(?x)` }, { `ub:name(?x,?n)` }}.

As described in Section 2.1, the conjunctive user query is then reformulated, optimized, translated into a single UNION SQL query and executed by PostgreSQL. Currently, Owlgres supports result set modifier operations (e.g. `FILTER`), although such operations are not pushed to the database but performed in memory as an operation on the result set. Some SPARQL constructs are currently not supported in Owlgres (e.g. `OPTIONAL`).

3 Owlgres in Action: Use Cases

In this section, we describe two use cases for Owlgres. As we argued in Section 1, the goal of Owlgres is to satisfy two needs. First, the need for a scalable and flexible persistent store for RDF and OWL data. Second, the need for standard reasoning services and efficient query answering over large knowledge bases.

3.1 DBpedia Infobox Data

The first use case is based on DBpedia, in particular the Infobox data, which includes the content of Wikipedia infoboxes (e.g. the content in the box on the right on the page for Albert Einstein[9]).

The DBpedia Infobox data contains roughly 24 million triples. The main problem with the DBpedia dataset is the unavailability of a proper TBox. The terminology is defined in multiple files which, for Owlgres, means some initial preprocessing. The ABox data has to be preprocessed too, especially because it is provided in N-TRIPLES format and the OWL API[10] used for the loader in Owlgres does not support N-TRIPLES. We used Sesame [4] to stream the N-TRIPLES into Owlgres and discovered a number of issues in the original dataset, in particular wrong datatype assertions.

We successfully loaded the DBpedia Infobox dataset with 2,198,649 concept assertions and 21,503,343 data role assertions on a Linux workstation with an Intel dual core 2 GHz and 2 GB of memory. Loading performance in Owlgres is roughly 2,000 statements per second. This is relatively slow. However, our focus

[9] http://en.wikipedia.org/wiki/Albert_Einstein
[10] http://owlapi.sourceforge.net/

for Owlgres v.0.1 was not on optimizing loading performance but on optimizing query performance. Little care has been given on analyzing loading performance. However, as Owlgres is a backward chaining reasoner, i.e. reasoning is performed at query time, we have great potential in getting good load performance (at least for the non-OWL *sameAs* schema).

The query we perform over DBpedia is for the book[11] *The Lord of the Rings*. As expected, both the DBpedia SPARQL endpoint[12] and Owlgres return the corresponding individual in the result set. In contrast, only Owlgres will return the individual on a query for the publication[13] *The Lord of the Rings*. The DBpedia SPARQL endpoint does not perform the inference required for this query.

3.2 POPS

POPS is an expertise location service that has been developed by Clark & Parsia for NASA with, as described by M. Grove in the W3C public use case,[14] the aim of "integrating NASA's information about its nearly 70,000 combined civil service and contractor workforce in one place, linking the relevant, related information to form a comprehensive data service for staffers, workforce planners, analysts, and related personnel". Currently, it is backed by Sesame and the data is managed in main memory. The dataset contains 10,494,493 assertions (621,546 concept assertions, 2,350,327 object role assertions and 7,522,620 data role assertions).

jSpace,[15] a visual query builder and generic RDF browser developed by Clark & Parsia, is used as the user front-end for the POPS data and we tested with Owlgres the initial query that is executed when jSpace is loaded on POPS. The reformulated set for the query without any optimizations resulted to be of size 3,276. It is obvious, that such a large number of SQL queries would make Owlgres practically useless.

The optimization that significantly minimizes the number of queries that have to be executed for this query is role simplification. By initially simplifying the conjunctive user query and subsequently each generated expansion, we reduce the reformulated set of queries to 9. By adding an inclusion axiom to the TBox, specifically an axiom that specifies the range for a POPS property, the reformulated set of queries can be optimized to one single query. This last query perfoms in 5,384 milliseconds.

4 Related Work

A number of implementation efforts in the Semantic Web community focus on efficient querying over RDF data. Triple stores like Sesame [4], Jena (SDB) [8]

[11] http://dbpedia.org/class/yago/Book106410904
[12] http://dbpedia.org/sparql
[13] http://dbpedia.org/class/yago/Publication106589574
[14] http://www.w3.org/2001/sw/sweo/public/UseCases/Nasa/
[15] http://www.clarkparsia.com/jspace/

and OpenLink Virtuoso [12] aim at efficient querying by translating SPARQL queries into SQL queries which are subsequently optimized and executed by the underlying RDBMS. Other projects focus on optimized native index structures [11, 18] or native persistent stores for RDF graphs.[16]

BigOWLIM [13] is a proprietary implementation that supports RDFS, OWL DLP and OWL Horst reasoning using a forward-chaining rule engine. In contrast, Owlgres supports DL-Lite$_R$ reasoning performed at query time.

Similar to Owlgres, QuOnto [1] is a querying system based on DL-Lite and has been used in particular for the integration of relational data [5]. Compared to Owlgres, QuOnto's focus is on integrating data from multiple relational database management systems while Owlgres manages RDF and OWL data as a knowledge base and provides querying as an inference service.

5 Conclusions and Future Work

We presented Owlgres, a scalable DL-Lite reasoner written for PostgreSQL and discussed the main features of DL-Lite in general. We described in details the query optimization techniques that have been implemented in Owlgres, optimizations which focus on minimizing the reformulated set of conjunctive queries and, therefore, the number of queries that are executed by PostgreSQL. We discussed the main aim of Owlgres with two use cases that show both goals of handling millions of assertions and reasoning over RDF and OWL data in the expressivity of DL-Lite$_R$.

Owlgres v.0.1 is a proof of concept. There are a number of directions on which we can focus in the future to improve Owlgres. First, tuning of PostgreSQL may help improving query performance even more and parallelisation may be useful to process more data. Second, a more optimized loading procedure may help to improve loading performance. Third, as Owlgres is not limited to PostgreSQL, supporting other RDBMS most likely would improve user acceptance of Owlgres in heterogeneous IT environments. Last but not least, improving the coverage of the SPARQL language is another important goal which most likely improves user acceptance.

[16] http://jena.hpl.hp.com/wiki/TDB

References

[1] Andrea Acciarri, Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Mattia Palmieri, and Riccardo Rosati. QuOnto: Querying ontologies. In *Proc. of the 20th Nat. Conf. on Artificial Intelligence (AAAI 2005)*, 2005.

[2] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language Reference. W3C Recommendation, 2004.

[3] Tim Berners-Lee. Linked Data: Design Issues. W3C, 2006.

[4] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF. In *International Semantic Web Conference 2002*, 2002.

[5] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. Efficient integration of relational data through DL ontologies. In *Proc. of the 20th International Workshop on Description Logics (DL-2007)*, 2007.

[6] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. In *Journal of Automated Reasoning*, 2007.

[7] Diego Calvanese, De Giacomo Giuseppe, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Linking Data to Ontologies: The Description Logic DL-Lite(A). In *OWLED-06*, 2006.

[8] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: implementing the semantic web recommendations. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 74–83, New York, NY, USA, 2004. ACM.

[9] Christiane Fellbaum. *WordNet An Electronic Lexical Database*. 1998.

[10] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. In *Web Semantics: Science, Services and Agents on the World Wide Web*, volume 3, pages 158–182, 2005.

[11] Andreas Harth, Jurgen Umbrich, Aidan Hogan, and Stefan Decker. YARS2: A Federated Repository for Searching and Querying Graph Structured Data. Technical report, 2007.

[12] Kingsley Idehen. A superplatform for merger driven data integration and business process services.

[13] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. OWLIM – a Pragmatic Semantic Repository for OWL. In *Proc. of Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2005), WISE 2005*.

[14] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 2004.

[15] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. *SIGMOD Rec.*, 14(2):256–276, 1984.

[16] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 2008.

[17] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A Core of Semantic Knowledge. In *16th international World Wide Web conference (WWW 2007)*, New York, NY, USA, 2007. ACM Press.

[18] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *Proc. of the 34th Intl Conf. on Very Large Data Bases (VLDB)*, 2008.