1st International Workshop on Data Engineering meets the Semantic Web (DESWeb)
In conjunction with ICDE 2010, 5-6 March 2010, Long Beach CA, USA

Optimized Data Access for Efficient Execution of Semantic Services

Thorsten Möller and Heiko Schuldt

Databases and Information Systems Group, Dept. of Computer Science, University of Basel, Switzerland
thorsten.moeller@unibas.ch, heiko.schuldt@unibas.ch

Abstract— **Executing Semantic Services requires, in contrast to traditional SOAP-based Web Services, frequent read and write accesses to graph-based semantic data stores – for instance, for the evaluation of preconditions or the materialization of service effects. Therefore, the overall performance of semantic service execution, in particular for composite services, is strongly affected by the efficiency of these reads and writes. In this paper we present two data access optimization techniques for semantic data stores: *Prepared Queries* and *Frame Caching*. The former reduces the costs for repeated query evaluation, e.g., in loops. The latter provides rapid access to frequently read triples or subgraphs based on materialized views using a Frame-based data structure. The described techniques have been implemented and evaluated on the basis of OSIRIS NEXT, our open infrastructure for Semantic Service support.**

I. INTRODUCTION

The proliferation of Service-oriented Computing has paved the way for a new computing paradigm that enables and facilitates the seamless interoperation of loosely coupled software components. Services as main abstraction provide functionality in a modular and well-defined way, based on standardized interfaces for description and invocation. In particular, Web service standards that have been devised in recent years cover a wide range of aspects such as service description and access (e.g., SOAP, WSDL), service combination which allows to define more sophisticated composite services (e.g., BPEL4WS), transaction support, and security and privacy measures. The vast success of service-oriented architectures has also brought forward Semantic Web Service standards such as OWL-S [1] and SAWSDL [2]. These standards enrich the purely syntactic description of (SOAP-based) Web services with additional details on the service's semantics, based on Description Logics (DL), in order to support the automated application-to-application interaction. This allows to address novel applications such as the automated semantics-based composition of services to ad hoc workflows, for instance in mobile Internet-based applications, and advanced failure handling strategies by dynamically choosing semantically equivalent or at least similar services to replace a service that has failed.

In this paper, we focus on efficient system support for the execution of composite semantic Web services described in OWL-S. In contrast to traditional workflow engines which orchestrate the execution of BPEL processes by invoking SOAP Web services in order in a request/reply style, thereby mainly acting as interpreters of BPEL process specifications, the execution of OWL-S processes is more complex. Semantic Web service profiles include preconditions and effects, expressed by means of logic formulas (e.g., decidable SRWL atoms or RIF production rules). Checking whether preconditions are satisfied w.r.t. a given knowledge base (KB) is done by translating those formulas into SPARQL [3] or SPARQL-DL [4] queries that are then evaluated against the KB. The KB is a specialized database for knowledge management that usually adopts a graph-based data model to seamlessly support RDF triples and that integrates inferencing engines for the purpose of automated deductive reasoning. In addition, the effects created as a result of service invocations need to be properly materialized in the KB. Therefore, efficient semantic service execution is strongly dependent on the efficient query (evaluation of preconditions) and update (materialization of effects) processing in the database(s) of the KB.

We apply two optimization techniques to semantic service execution: *Prepared Queries*, which are well known from databases, and *Frame Caching*. The former reduces the costs for repeated query transformation (e.g., for the evaluation of service preconditions) and does also reduce the number of KB updates, which in turn improves the performance of queries that involve inferencing. The latter aims at reducing the number of KB queries by keeping materialized views of frequently accessed individuals and data values in Frame-based data structures [5]. The proposed techniques were evaluated in detail on the basis of our OWL-S execution engine which is part of OSIRIS Next[1]. OSIRIS Next provides flexible and dynamic failure recovery of composite Semantic Web services [6] and executes these composite services in a fully distributed way [7]. The evaluations show a performance boost for repeated service execution and for execution of composite services consisting of loops and iterative control constructs by at least one order of magnitude, even when the KB itself is completely kept in main memory. The performance gain can be even much higher depending on the structure of the KB, especially w.r.t. the topology of the RDF graph.

We note that there is a huge body of work on providing efficient data access contributed by the database and knowledge representation communities over the last decades (e.g. [8], [9], [10]). This includes (i) query answering using views, (ii)

[1]Available as open-source project at http://on.cs.unibas.ch/.

algebraic rewriting and reordering of execution plans, (iii) result caching, (iv) indexing, (v) storage models, and (vi) programming interfaces. Moreover, with the increasing use of DLs, also optimizations to (vii) logic based query answering have been proposed. The techniques proposed in this paper rely on and contribute back to (iii) and (vi), applied to semantic service execution. We do neither address the efficiency of the reasoning services, nor propose novel methods falling into the other categories. However, in contrast to work done in the past, the results of the proposed approach are a valuable contribution to the recent research towards main memory data storage systems [11], since we can show reasonable performance gains especially for main memory KBs.

The remainder of the paper is organized as follows. In Section II we briefly introduce OWL-S and the execution task for composite services. Section III introduces the two performance optimization techniques. The utility of both techniques was assessed by a quantitative experimental evaluation, which we present in Section IV. Section V concludes.

II. Execution of OWL-S Services

In short, OWL-S (formerly known as DAML-S) is a framework for describing (i) functional and non-functional properties of services, (ii) compositions of services in terms of a process model, and (iii) the technical details on how they can be invoked. Its formal specification comes as a set of layered OWL-based ontologies, consisting of the *Service Profile*, *Process Model*, and *Grounding*. The service profile presents *what the service does* in terms of *inputs* consumed, *outputs* produced, *preconditions* that need to be satisfied so that it can perform when invoked, and *effects* created upon termination. Inputs and outputs can be thought as variables that are typed either to OWL concepts or OWL data ranges, while preconditions and effects can be expressed by means of logic formulas. The process model describes *how a service works*. Services are classified as either (i) *atomic* or (ii) *composite* (services that can be further decomposed into atomic or other more fine-grained composite services). The process model supports the orchestration of composite services by means of various collection, conditional, and loop control constructs such as Sequence, Split-Join, If-Then-Else, Repeat-While etc. In other words, the process model specifies control and data flow. Both, the service profile and process model are thought of as descriptions of abstract services. Groundings provide the necessary details for services to be actually usable, in terms of message format, transport protocol, and addressing. This allows service consumers to actually invoke them. Several different types of groundings have been devised, amongst of which the WSDL grounding enables integration of standard Web Services.

Service execution, in general, comprises all the activities that need to be carried out at runtime in order to work off the control and data flow of composite services in a correct, consistent, and reliable manner – like in classical workflow or process management. Usually, this task is handled by dedicated execution engines acting on behalf of a client.

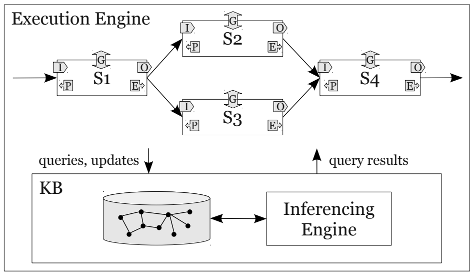

Fig. 1. Overview composite semantic service execution.

OWL-S builds on OWL and RDF. Data is represented in graph-based models, which are either maintained entirely in main memory, or (at least partially) in persistent triple stores. Moreover, such models usually realize the notion of knowledge bases. In short, they provide a unified and monolithic view on schema (a.k.a. TBox) as well as instance (a.k.a. ABox) data and support standard DL inferencing tasks. Applied to the context of semantic service execution this means that a KB contains (i) required domain and OWL-S ontologies (TBox), (ii) pre-existing domain individuals and data values, (iii) individuals and data values created in the course of execution, and (iv) the service description(s) themselves; the latter three all falling into the ABox.

Carrying out service execution by an engine is characterized by recurring sub tasks, depicted in Figure 1:

- Read the control and data flow constructs from the process model so that the engine can interpret and execute them (according to their operational semantics).
- Read preconditions (P) of services from the KB and check if they are satisfied w.r.t. to the current state of the KB.
- Read inputs (I) for service invocations from the KB. Write outputs (O) produced by service invocations (intermediate results) back to the KB so that they are available for subsequent use.
- Materialize effects (E) as a result of service invocations in the KB to correctly represent the current execution state.
- Read service grounding details (G) from the KB in order to prepare service invocation messages and process replies.

All this tasks involve queries against and/or updates to the associated KB. In fact, the KB needs to be queried and updated almost permanently. As a consequence, the overall performance of service execution is dominated to a large extent by the runtime efficiency of basic operations and reasoning services offered by the KB implementation.

III. Optimization Techniques

In this section we will introduce two optimization techniques that we have developed to improve efficiency of service execution, both in terms of execution time and resource utilization.

A. Prepared Queries

Service preconditions and conditions used in conditional control constructs are usually expressed using DL-safe SWRL atoms (note, however, that OWL-S does not mandate the use

TABLE I

SWRL ATOMS, THEIR SEMANTICS, AND MAPPING TO SPARQL BGP

Abstract Syntax	**Semantics** / **Triple form for SPARQL BGP**
Class(t, c)	$(\mu(t))^{\mathcal{I}} \in c^{\mathcal{I}}$
	$\langle a, \text{rdf:type}, c \rangle$
IndividualProperty(t, p, u)	$\langle (\mu(t))^{\mathcal{I}}, (\mu(u))^{\mathcal{I}} \rangle \in p^{\mathcal{I}}$
	$\langle \mu(t), \text{p}, \mu(u) \rangle$
DataProperty(t, p, v)	$\langle (\mu(t))^{\mathcal{I}}, (\mu(v))^{\mathcal{D}} \rangle \in p^{\mathcal{I}}$
	$\langle \mu(t), \text{p}, \mu(u) \rangle$
SameIndividual(t_1, t_2)	$(\mu(t_1))^{\mathcal{I}} = (\mu(t_2))^{\mathcal{I}}$
	$\langle \mu(t_1), \text{owl:sameAs}, \mu(t_2) \rangle$
DifferentIndividuals(t_1, t_2)	$(\mu(t_1))^{\mathcal{I}} \neq (\mu(t_2))^{\mathcal{I}}$
	$\langle \mu(t_1), \text{owl:differentFrom}, \mu(t_2) \rangle$

Using standard interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \Delta^{\mathcal{D}}, \cdot^{\mathcal{I}})$; c a concept; p a object or data property; t, u a individual or individual variable; v a data value or data variable.

nor support of particular formalisms). More formally, a SWRL (pre-) condition p is a (possibly empty) conjunction of SWRL atoms $p = a_1 \wedge \ldots \wedge a_n$ $(n \geq 0)$. Atoms can be either of the form as illustrated in Table I, or from the set of built-in SWRL atoms. The latter represent basic functions; due to space restrictions not listed here. Atoms have arguments called terms $a(t_1, \ldots t_m)$, where t can be either a variable or a constant symbol; the latter referring to some individual, property, or concept in the KB. An atom is *ground* if it does not contain variables. Note that a variable always refers to a process variable declared in the process model of the corresponding service, that is, an input, local, or existential variable. Inputs and locals are in fact always bound to a value at runtime, whereas existentials will be bound only by evaluating a condition. This means that before condition evaluation atoms can be made ground if they contain only input or local variables, but remain partially-ground if they contain existentials. We define a mapping μ to be the identity for constant symbols and for variables the current value bound.

$$\mu(t) = \begin{cases} t & t \text{ is a constant symbol,} \\ val(t) & t \text{ is a variable.} \end{cases}$$

Finally, a condition is satisfied (evaluates to *true*) iff each atom a_i is entailed by the KB, that is, $\forall a_i \in p\colon KB \models a_i$; and the empty precondition being trivially satisfied.

The approach used to check if a (pre-) condition is satisfied is to translate it to a SPARQL or SPARQL-DL query and evaluate it against the current state of the KB. This is possible because the formal semantics of a conjunction of SWRL atoms can be preserved when translating them to basic graph patterns (BGP). Table I outlines how SWRL atoms can be represented as BGPs. Finally, a condition is *uniquely* satisfied if evaluation of the translated query yields exactly one result, it is not uniquely satisfied if there is more than one result, and it is not satisfied for no result. For example, the following condition in abstract syntax

$$\text{Class}(x, InsuredPerson) \wedge \text{Class}(x, PhysicalObject)$$

where x shall represent a process input, its value shall refer some individual in the domain, e.g., $\mu(x) = Bob$, and

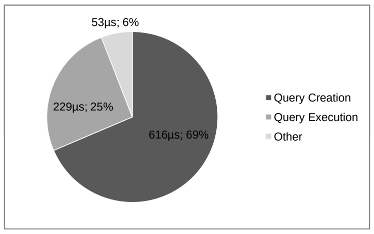

Fig. 2. Ratio of query creation and query execution on total query evaluation time (in microseconds) for conventional approach using *Repeat-While* service.

$InsuredPerson$, $PhysicalObject$ shall be classes of some domain ontology, would be translated to

```
SELECT *
WHERE { :Bob rdf:type :InsuredPerson ;
             rdf:type :PhysicalObject . }
```

The condition is satisfied if evaluation of the query against the KB has a result (which would be emtpy in this case as there is no variable to project to).

The conventional approach to condition evaluation starts with replacing each atom which contains input and/or local variables by a new atom where such variables have been substituted by their value, i.e., creating a possibly ground atom. Since conditions are in the majority of cases expressed using input variables, this implies additional work in all of these cases and, nota bene, an insertion of new triples in the KB, which is required for the subsequent translation to a query. Also recall that SWRL conditions are part of the service description, hence, they are also represented as RDF triples. While those insertions are cheap w.r.t. to the KB's database, they have severe consequences for the reasoning engine integrated into the KB (which is usually present as reasoning is basically inevitable). Unfortunately, today's reasoning engines do not yet perform well under (frequent) KB updates since they need to exhaustively re-perform consistency checks, classifications, and realizations. The consequence is that such updates to the database of the KB provoke (more or less) high delays for subsequent queries, thus, reducing the overall performance.

Yet there is another weakness when it comes to repeated evaluation of the same condition, for instance when a conditional control construct in a composite service is executed multiple times (e.g., as part of a loop). Figure 2 shows the result of an analysis to break down the times of overall condition evaluation using a Repeat-While control construct having a single *lessThan* built-in SWRL atom in the condition. It shows that for repeated evaluation of this condition a considerable amount of time is spent just for the creation of the query (create ground atom and translate to SPARQL), thereby exceeding the time for the actual evaluation of the query against the KB by a factor greater than two.

An approach to optimize condition evaluation thus has to address those problems by factoring out the efforts induced by the query creation process. The proposed *prepared queries* simplify and optimize the process of condition checking by avoiding the creation of new ground atoms (from unground atoms by variable binding) in the KB. This is done in three

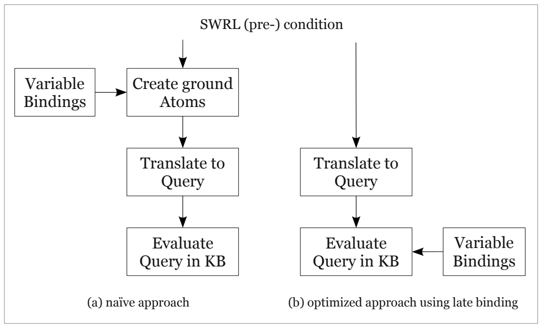

Fig. 3. Comparison of (pre-) condition evaluation procedure for conventional and optimized approach using prepared queries.

steps. First, we need to provide the possibility of translating unground atoms directly to a SPARQL/SPARQL-DL query. Second, we allow for late binding of variables occurring in the query at query execution time. Third, having late binding of variables gives the possibility of reusing queries, that is, translating a query just once and evaluating it as often as needed, thus also supporting conditions that need to be checked multiple times.

These three steps can be realized by prepared queries which are kept in memory as long as they are required. A prepared query represents a (pre-compiled) SPARQL/SPARQL-DL statement and allows the late binding of variables when the query is actually evaluated (executed). As such, prepared queries are similar to prepared statements, a well known abstraction provided by today's relational DBMS access interfaces (e.g., in JDBC), aiming for similar improvements of efficiency. However, implementations of prepared queries are currently not yet available in well known frameworks such as Sesame and the OWL-API. Only Jena/ARQ already provides some support for late binding of variables. Since our OWL-S execution engine builds on Jena, we have extended the engine to allow for reusing queries and for better abstracting late binding by the programming interface. This yields a simplified (pre-) condition checking process for OWL-S as depicted in Figure 3. Yet its most important advantage is that insertion of additional ground atoms in the KB is eliminated, thus, preventing that reasoning engines need to (exhaustively) re-check consistency, re-classify, and re-realize the KB.

B. Frame Caching

A second challenge is to increase the performance of (i) semantic services which may have to be executed repeatedly and (ii) composite services which consist of loop control constructs (e.g. Repeat-While or For-Each) which, at execution time, repeatedly loop over the body. In both cases, the same information of the process model (i.e., control and data flow) and the grounding specification is required by the engine for each repetition. If an execution engine is not aware of past query results from accessing the KB, it will issue the same query over and over again. These kinds of reads can be optimized by keeping the result of the first query in a cache and by re-using this result, if appropriate, for subsequent reads. However, finding an appropriate caching solution in the context of semantic service execution is not straightforward as it has to take into account the following aspects:

- Locality of cached data: Either close to the KB (probably inside the KB) vs. close to the application which acts as client to the KB and which uses the query results.
- Granularity of cached data: Limiting cached data to exactly the query result vs. more advanced look-ahead strategies where data that is likely to be read in the future is prefetched and cached in advance.
- Representation of cached data: Graph-based, essentially in the same representation as stored the KB vs. representation in other data structures that may be more suited w.r.t. data access patterns of the client application.
- Cache coherence: Invalidation of cached data in the presence of updates to the original data in the KB due to concurrent access by multiple clients.
- Implicit data which is not materialized in the KB but inferred by reasoning engines (dynamically at query time).

The *frame caching* approach we have developed addresses all the issues summarized above and provides (i) materialized views of sets of proximate triples or sub-graphs of a KB, using (ii) frame-based data structures, which, at the same time, also realize (iii) a simple form of a look-ahead cache, (iv) local to the place where they are used, and (v) possibly contain inferred data.

The notion of *frame* was introduced in frame-based systems [5] as an alternative to logic-oriented knowledge representation systems. More formally, a frame F contains a set of *slots* $s_1, \ldots, s_n$, similar to entries in a record. Slot *fillers* represent the value of a slot which can be data values or again frames, thereby allowing nested frames. It is possible to use frames for having a record-like view to triples $\langle s, *^p, *^o \rangle$ in the KB, where the subject s corresponds to the frame F, properties ($*^p$) correspond to slots, and objects ($*^o$) to their fillers. If fillers are again frames, one can represent sub-graphs by nested frames. A frame will always be created from the results returned by KB read operations, i.e., the selected triples or sub-graphs respectively. Using OO languages, frames can be easily represented by objects (possibly having no methods, i.e., behavior). In doing so, one will get rapid access to the fillers of a frame. Consequently, one can have rapid access to (all) objects of some subject once a corresponding frame was filled. The performance evaluation which will be discussed in detail in the following section shows that access is still faster than directly fetching triples from a KB even if the KB is entirely kept in main memory.

In the context of OWL-S execution we use frame caching to gain rapid access to the process model (i.e., control and data flow) and to groundings. Repeated execution of the same service, or of loop control constructs that iterate over their body, will profit from having instant access to them compared to repeatedly fetching them from the KB. The same applies for groundings when services are repeatedly invoked. The entire process model of some service or a grounding can be kept by one nested frame, which provides a concise representation.

Frames may also include inferred data which does not need to be recomputed. Frames come with moderate additional memory requirements as slots are basically references to data values or other frames.

Representation of the process model by a frame is highly beneficial if the former is not subject to modifications at execution time. However, highly dynamic application scenarios where adaptation at execution time is required has the consequence that frames need to be invalidated; at best partially only for those parts which were modified.

IV. Experimental Evaluation

We have implemented prepared queries and frame caching in OSIRIS Next, our OWL-S execution engine. The implementation allows both to be toggled on and off; subsequently we will refer to optimized (on) versus conventional (off) configuration. This provides larger flexibility and allows us to easily quantify the utility of both in any constellation, i.e., using any kind of invocable services and using differently sized and shaped KBs. We have created various service descriptions, designed specifically for testing and benchmarking purposes. Because of their specific design, they cover a broad range of possible OWL-S service descriptions. Hence, they are well suited to simulate different characteristic cases. More precisely, we used the following services[2].

- *Any-Order.* Uses the Any-Order control construct of OWL-S consisting of three elements, each being the same atomic service that logs a given message. Does not contain preconditions.
- *For-Each.* Uses a For-Each control construct that contains an atomic service which plays a given MIDI note. This is used to play an input list of MIDI notes. Uses a consistency check precondition that asserts that each note can actually be played.
- *If-Then-Else (1 and 2).* Two services, each consisting of a If-Then-Else control construct, with different branching conditions.
- *Repeat-While.* Uses a Repeat-While control construct with a simple loop condition. The service does nothing but an increment to a given input number until a target value is reached.
- *Translator.* Atomic service realizing language translation of words. Uses a precondition that asserts that source and target language are supported.
- *JavaGrounding.* Atomic service that does a simple power computation of two numbers. Does not have a precondition.

All experiments were conducted on commodity hardware (Win XP, 3.4 GHz Intel Pentium D 32bit, 2GB RAM, Java 6, max. Java heap size ∼1.5GB). We used Pellet [12] as the reasoner attached to the KB. Note that in all tests the KB was kept entirely in RAM. All tests started from an already populated KB where a consistency check, classification, and realization was initially done. In a first round we executed all services in the conventional and the optimized setting to measure the absolute speedup of using prepared queries and frame caching. Each test run was repeated ten times and the average time was taken. Table II shows results of this evaluation.

[2]Available via download from the OSIRIS Next website.

TABLE II
Execution speedup of exemplary test cases

Test Case	Conventional	Optimized	Speedup
Any-Order	$40ms$	$39ms$	1.02
For-Each	$5410ms$	$448ms$	12.08
If-Then-Else 1	$180ms$	$33ms$	5.45
If-Then-Else 2	$27ms$	$3ms$	12.33
Java Grounding	$502\mu s$	$42\mu s$	11.96
Repeat-While	$2834ms$	$828ms$	3.42
Translator	$6625ms$	$3631ms$	1.82

The execution time of the *Any-Order* service can not be improved because it neither has a precondition nor is any part of the process model accessed more than once. We decided to include this test in order to analyze if creation of frame cache objects introduces a considerable overhead. As the values show, the execution times are basically the same in both settings. A more fine grained analysis has shown that creation of the frame that represents the entire grounding (having 8 slots) takes 80ns on average. This shows that creation of frames is a cheap operation and its overhead can be almost neglected. The *For-Each* test is interesting as the speedup is solely caused by the optimized condition evaluation process, that is, the elimination of additional KB inserts. In the conventional implementation the execution time is dominated by the need of repeated classification and realization by the reasoner after a KB update. The *JavaGrounding* test run was executing the service 3000 times and the average value for one execution was taken. For the *Translator* service we measured just the speedup of precondition evaluation (i.e., not including execution time).

Figure 4 depicts the execution times measured for condition evaluation and actual process execution for the *Repeat-While* service. Apart from the reduction of execution time due to frame caching, the numbers for condition evaluation show that with prepared queries the overhead of repeated translation from SWRL atoms to queries is basically eliminated (see also Figure 2).

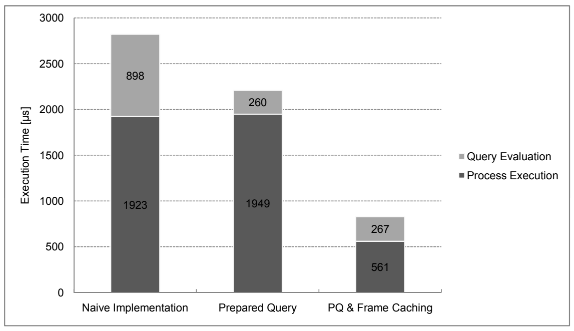

Fig. 4. Composition of total execution times (in microseconds) for Repeat-Until service using conventional and optimized engine configurations.

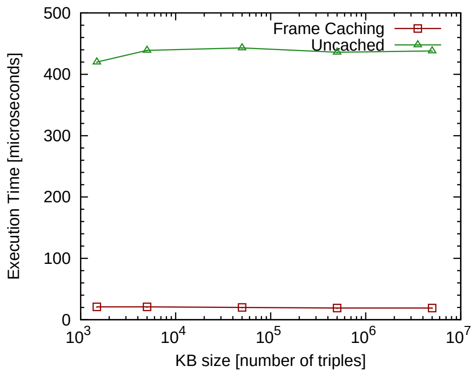

Fig. 5. Interrelation of Execution Time and KB size – OWL-S Services

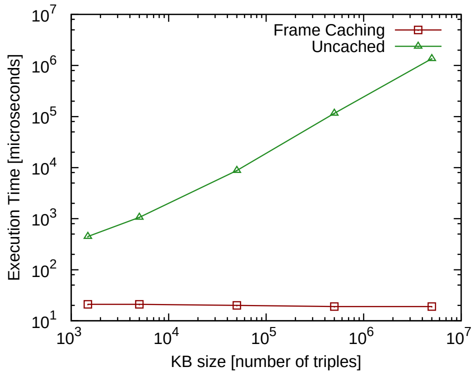

Fig. 6. Interrelation of Execution Time and KB size – LUBM Individuals

So far, all tests were conducted with a "minimal" KB containing not more data than actually required. In a second round we repeated the *JavaGrounding* test run with incrementally growing KB (ABox). In one test we enlarged it by additional synthetically generated OWL-S service descriptions, while in another test we enlarged it by randomly generated individuals and assertions using the LUBM ontology [13]. These tests have been made to confirm our expectation that execution performance using frame caching should be independent of the KB size and its structure. Note that the *JavaGrounding* service is designed not to involve reasoning, thus, times can not be distorted by reasoning.

Figures 5 and 6 show that execution time remains constant when using frame caching, even with increasing KB sizes. However, the numbers for the uncached execution feature significant differences compared to the cached case. Whereas in case of adding more OWL-S descriptions (Figure 5) execution time also remains constant—but at a higher level compared to the cached case—it increases linearly with the KB size in the other case. This can be explained by looking at how indexing is designed for graph-based data structures in Jena. In case of adding more OWL-S services the KB is enlarged by adding "new assertions about *new* individuals". This ensures that the number of triples $|\langle s, *^p, *^o \rangle|$ for any subject s remains constant with KB increase, which results in constant access times provided by the index. In the LUBM test, however, the KB is enlarged by adding "new assertions about *existing* individuals", that is, $|\langle s, *^p, *^o \rangle|$ for any subject s is proportional to KB increase, which represents a topology of the KB resulting in the highest efforts for reads.

V. Discussion and Future Work

In this paper, we have introduced two approaches to speed-up the execution of (composite) semantic services. First, result caching of KB queries. In addition to the service execution tasks, this technique is generally applicable to applications that need to frequently fetch the same triples or sub graphs, and even outperforms direct read access on KBs which are entirely kept in main memory. Second, we have presented prepared queries that allow for late binding of variables at query evaluation time and for efficient re-evaluation.

The experiments with different KB sizes and structures have indicated further optimization potential, not only w.r.t. query evaluation and optimization but also on the physical organization of graph-based data. Conventionally, a KB is considered as one coherent unit on the logical level and often centrally stored at physical layer, with many efforts that focus on scalability issues of large monolithic KBs. Having more knowledge about KBs, for instance, by means of runtime metrics capturing structural properties or access patterns, would allow to better organize or partition KBs. An optimized structure may allow for (temporarily) masking data which is irrelevant for certain reasoning tasks, to better parallelize reasoning tasks, or to improve performance under frequent updates. We plan to address these aspects in future work.

Acknowledgment

This work has been funded by the Hasler Foundation within the research project LoCa.

References

[1] OWL-S Services Coalition. (2004) OWL-S: Semantic Markup for Web Services. [Online]. Available: http://www.w3.org/Submission/OWL-S
[2] J. Farrell and H. Lausen. (2007) Semantic Annotations for WSDL and XML Schema. [Online]. Available: http://www.w3.org/TR/sawsdl
[3] The SPARQL Working Group. (2008) SPARQL Query Language for RDF. [Online]. Available: http://www.w3.org/TR/rdf-sparql-query
[4] E. Sirin and B. Parsia, "SPARQL-DL: SPARQL Query for OWL-DL," in *3rd OWL: Experiences and Directions Workshop (OWLED2007)*, 2007.
[5] M. Minski, *Mind Design*. MIT Press, 1981, ch. A Framework for Representing Knowledge.
[6] T. Möller and H. Schuldt, "Control Flow Intervention for Semantic Failure Handling during Composite Serice Execution," in *Proc. ICWS'08*, 2008, pp. 834–835.
[7] T. Möller and H. Schuldt, "A Platform to Support Decentralized and Dynamically Distributed P2P Composite OWL-S Service Execution," in *Proc. MW4SOC'07*. ACM, 2007.
[8] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database System implementation*. Prentice-Hall, 2000.
[9] P. C. Lockemann, H.-H. Nagel, and I. M. Walter, "Databases for Knowledge Bases: Empirical Study of a Knowledge Base Management System for a Semantic Network," *Data Knowl. Eng.*, vol. 7, pp. 115–154, 1991.
[10] J. Mylopoulos and M. L. Brodie, "Knowledge Bases and Databases: Current Trends and Future Directions," in *1st Workshop on Information Systems and Artifficial Intelligence*, 1990, pp. 153–180.
[11] M. Stonebraker et al, "The End of an Architectural Era (It's Time for a Complete Rewrite)," in *Proc. 33rd VLDB*, 2007, pp. 1150–1160.
[12] Clark & Parsia, LLC. Pellet: The Open Source OWL Reasoner. [Online]. Available: http://clarkparsia.com/pellet
[13] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A Benchmark for OWL Knowledge Base Systems," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, no. 2-3, pp. 158–182, October 2005.