Developing Efficient Ontology-based Systems Using A-box Data from Running Applications

Heiko Paulheim[1] and Lars Meyer[2]

[1] Technische Universität Darmstadt
Knowledge Engineering Group
`paulheim@ke.tu-darmstadt.de`
[2] SAP Research
{`lars.meyer`}`@sap.com`

Abstract. Today, information is typically contained in different IT systems. Ontologies and semantic integration have been proposed for integrating this information with the purpose of meaningful processing and providing useful services on top of that information. As applications, at the same time, modify the information during run-time, the information to be integrated has a dynamic nature. In this paper, we discuss performance aspects of integrating such dynamically changing A-box data from running applications, point out several technical alternatives, and present performance measures for those alternatives. We show how the findings are applied in two different application examples.

1 Introduction

In a typical software landscape, information is contained in different systems – databases, legacy systems, desktop applications, and so forth. As users have to work with information from those different systems, integration is required. For providing a meaningful, semantic integration, ontologies have been widely discussed. There are several possible utilizations of such an integration, each providing different benefits to end users [1]:

- On a semantic desktop, novel ways of searching for data in applications are made possible by extracting semantically annotated information from applications [2, 3].
- User interfaces can be automatically adapted according to the users' needs by having a reasoner analyze the UI components, the data they contain, and the user's needs [4].
- Help on applications can be provided at run-time, adapted according to the system's current state and/or a user model [5, 6].
- Software components can be automatically integrated by having a reasoner process events raised by different components and thereby coordinating user interactions across different, heterogeneous components [7].
- Interactive, ontology-based visualizations of the information contained in different related applications can assist the user in fulfilling knowledge-intensive tasks [8].

In all of those cases, the information is contained in *running* applications, which means that it is highly *dynamic* and thus needs to be integrated at run-time. Furthermore, *reasoning* on that data is essential for providing valuable information to the end user. Thus, it is required that a reasoner has efficient access to the data as its A-box. At the same time, user interaction is involved all of the cases, which imposes strict requirements in terms of performance. Thus, high performance mechanisms for reasoning on dynamic data from running applications are needed.

With this paper, we investigate different architectural alternatives of building a systems which support efficient integration and reasoning about running software applications, and we analyze the performance impact of the different alternatives with respect to *dynamic* data. In two examples, we show how the findings can be applied to improve the performance of real-world applications of semantic integration.

The rest of this paper is structured as follows. In section 2, we introduce our basic reasoning framework. Section 3 discusses different approaches for optimization, which are evaluated in section 4. In section 5, we introduce two example use cases for our framework and discuss how they benefit from the optimization strategies. We conclude with a review on related work, a summary, and an outlook on future work.

2 Basic Architecture

From our work in application integration, we have derived a *generic architecture* for reasoning on running software applications. In [9], we have analyzed two different basic architectural variants for providing A-box data from running software components to a reasoner:

1. In a *pushing* approach, software components inform the reasoner about updates, and the reasoner keeps an up-to-date representation in its own A-box, which duplicates the original data contained in the different components.
2. In a *pulling* approach, the reasoner does not maintain an A-box. Instead, it dynamically pulls instance data from the components whenever the evaluation of a query demands for that instance data.

Our experiments have shown that only pulling approaches are feasible for building a scalable solution [9]. The reason is that given a highly dynamic component which changes its state quite frequently, the reasoner is kept busy with processing the updates on its A-box. Once the update frequency exceeds a certain threshold, the reasoner is overloaded and cannot answer any queries anymore. On the other hand, many of those updates are unnecessary, e.g., if the same facts are overwritten many times without being used in a query. A pulling approach avoids those unnecessary updates, which only create system load without any benefit.

Furthermore, maintaining an A-box with information which is also kept in the system leads to an overhead due to double bookkeeping and can therefore

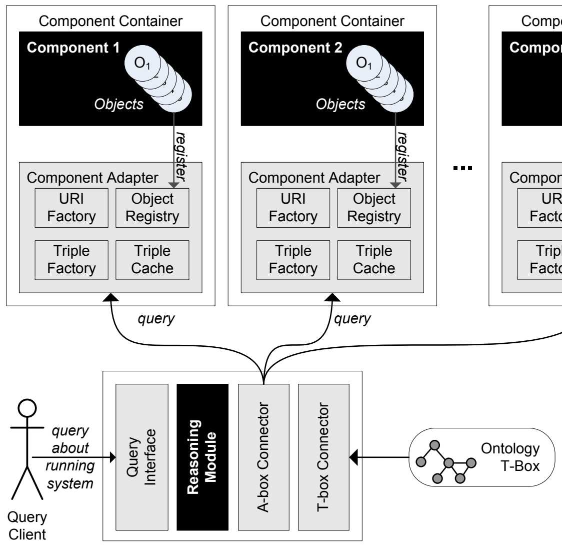


Fig. 1. General architecture of our framework for reasoning about running applications.

cause consistency problems: with a pushing approach, the updates and queries need to be properly queued in order to guarantee correct answers to each query.

Fig. 1 shows the basic building blocks of our framework. It depicts a number of software components which are to be integrated. Each component (which we treat as a black box) is encapsulated in a container, which provides an adapter to the reasoner. The adapter consists of four essential parts:

– Integrated components create and hold objects, which they register at the adapter's *object registry* to reveal them to the reasoner. Each component may process different types of objects, and each type of object may be processed by different components. There are different possible strategies of informing the registry of updates of the registered objects (e.g., using the observer pattern or change listeners); in our current implementation, the component actively sends updates to the object registry.
– From those objects, a *triple factory* creates data that is usable by the reasoner, i.e., RDF triples. The triple factory is also responsible for processing mappings between the component's class model and the ontology used for information exchange. Those mappings are defined in a flexible, rule-based language, which is also applicable for conceptually heterogeneous class models [10].
– To create such triples, a URI is needed for each object. The *URI factory* is responsible for creating such URIs which are unambiguous and unique throughout the whole integrated system.
– To improve performance, triples may be cached by the component. There are different variants such as lazy or eager caches, which will we analyze in more detail in the subsequent sections.

These adapters are used by the reasoner's A-box connector to dynamically resolve queries. In addition to that A-box connector, a T-box connector provides the T-box part of the ontology (i.e., the definition of classes and relations) used as a common ground for integrating information from the different components. In contrast to the A-box, the T-box is considered as static, and the T-box connector loads it once when the system starts.

The reasoner has a query interface that allows client components to pose queries about the running system. Client components may be, e.g., internal components, such as an event processing logic, as discussed in section 5.1, or a graphical user interface providing an endpoint for querying the system, as discussed in section 5.2. The query interface may use languages such as SPARQL or F-Logic. For the prototype implementation of our framework, we have used OntoBroker [11] as a reasoner, and F-Logic [12] as a query language.

When a query runs, the results of that query have to be consistent. Thus, updates occuring between the start of a query and its end should not be considered when computing the result. To provide such a consistent query answering mechanism, the reasoner sends a lock signal to the component wrappers when a query is started. Updates coming from the component are then queued until the reasoner notifies the component that the query has been finished.

3 Aspects of Optimization

In the previous section, we have sketched the basic architecture of our system. There are different variants of implementing that architecture. In [9], we have discussed two basic aspects: centralized and decentralized processing, and using a redundant A-box vs. using connectors for retrieving instance data at query time. These results led to the architecture introduced in Sect. 2, using a *centralized reasoner* and *connectors for retrieving A-box data*. In this section, we have a closer look at two design aspects which allow several variations: the design of the rules which make the reasoner invoke the A-box connector, and the use of caches.

3.1 Design of Connector Invocation Rules

To make the reasoner invoke a connector, a rule is needed whose head indicates the type of information the connector will deliver, and whose body contains a statement for actually calling that connector. Technically, a connector is wired to the reasoner with a predicate. For example, a connector providing instance data for an object can be addressed with a **generic rule** as follows[1]:

$$instance\_connector(?I, ?C) \Rightarrow ?C(?I). \qquad (1)$$

[1] We use the common SWRL human readable syntax for rules, although in SWRL, variables are not allowed for predicates. In our implemented prototype, we have used F-Logic for formulating the rules, which allows for using variables for predicates.

The reasoning framework, OntoBroker in our case, is responsible for dispatching the use of the predicate *instance_connector* to an implementation of that connector, i.e. a Java method. This method then provides a set of bindings for the variables (in this case: $?I$ and $?C$). If some of the variables are already bound, the contract is that the method returns the valid bindings for the unbound variables which yield a true statement given the already bound ones. Consider the following example query, asking for all instances of a class `#Person`:

```
SELECT ?I WHERE { ?I rdf:type #Person }
```

The resolution of this query leads to the invocation of rule 1 with the variable $?C$ bound to `#Person`. The connector method now returns a set of bindings for the variable $?I$ for which the statement $\#Person(?I)$ is true. The reasoner then substitutes the results in the query and returns the result.

The mechanism defined in rule 1 is the most basic way of integrating a connector which delivers information about instances and the classes they belong to. As it has to be evaluated in each condition in a rule's body where statements like $?C(?I)$ occur (either unbound or with one or both of the variables bound), the connector is invoked very frequently. Since invoking a connector may be a relatively costly operation (even with caches involved, as described below), this is a solution which may imply some performance issues.

A possible refinement is the use of additional constraints. The idea is that for each integrated software component, the set of possible ontology classes the data objects may belong to is known. Given that the union of those sets over all components is `#Class1` through `#ClassN`, the above rule can be refined to an **extended rule**:

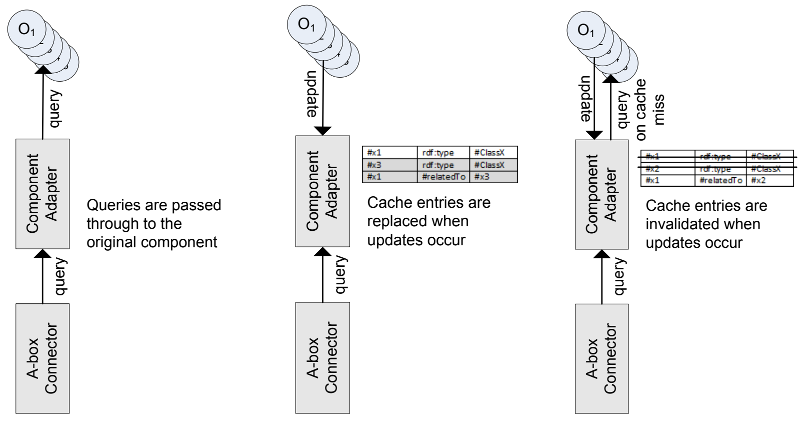
$$\begin{aligned} &(equal(?C, \#Class1) \vee equal(?C, \#Class2) ... \vee equal(?C, \#ClassN)) \\ \wedge\ &instance\_connector(?I, ?C) \\ \Rightarrow\ &?C(?I) \end{aligned} \tag{2}$$

Assuming a left to right order of evaluation of the rule's body, the connector is now only invoked in cases where the variable $?C$ is bound to one of the given values. Therefore, the number of the connector's invocations can be drastically reduced.

A variant of that solution is the use of **single rules** instead of one large rule:

$$\begin{aligned} instance\_connector(?I, \#Class1) &\Rightarrow \#Class1(?I) \\ instance\_connector(?I, \#Class2) &\Rightarrow \#Class2(?I) \\ &... \\ instance\_connector(?I, \#ClassN) &\Rightarrow \#ClassN(?I) \end{aligned} \tag{3}$$

In that case, the connector is not always invoked when evaluating a statement of type $?C(?I)$. Instead, each rule is only invoked for exactly one binding of $?C$. In the example query above, only the first rule's body would be evaluated at all, invoking the connector once with one bound variable. On the other hand, the number of rules the reasoner has to evaluate for answering a query is increased.



(a) Without cache (b) Eager cache (c) Lazy cache

Fig. 2. Different variants for using caches

The above example rules show how to invoke the *instance_connector* wrapper, which returns statements about category membership of instances. The other important is $relation\_connector(?X, ?R, ?Y)$, which has three variables. It returns the set of all triples where object $?X$ is in relation $?R$ with object Y. As for the *instance_connector* wrapper, the corresponding types of invocation rules exist.

For this paper, we have analyzed the performance impact of all three rule types: the generic rule (1), the use of an extended rule (2), and the use of single rules (3).

3.2 Distributed Caching of A-Box Fragments

To speed up the answer of our connectors, partly caching instance data in the connector is a good strategy [13], although it slightly contradicts to the idea of avoiding double bookkeeping – it is the classic trade-off of redundancy vs. performance. We have analyzed three different variants: using no caches at all, i.e., each query for instance data is directly passed to the underlying objects, and statements are assembled at query time (see Fig. 2(a)); and using *eager* and *lazy* caching. While the eager cache updates the required statements for each object when that object changes (see Fig. 2(b)), the lazy cache flags statements as invalid upon change of the represented object, and re-creates them only if they are requested (see Fig. 2(c)).

While using no caches at all avoids unnecessary workload when an update occurs, eager caches are supposed to be the fastest to respond to queries. Lazy caches can provide a compromise between the two, allowing fast responses to queries as well as avoiding unnecessary workload. In the next section, we will analyze those effects in more detail.

	Description	Query in SPARQL
1	Get all objects of type C	`SELECT ?I WHERE {?I rdf:type C.}`

2a
2b
2a+b
	Get all objects of type C in a relation R with object O	`SELECT ?I WHERE {?I rdf:type C. ?I R O.}`
		`SELECT ?I WHERE {?I rdf:type C. O R ?I.}`
		`SELECT ?I WHERE {{?I rdf:type C. ?I R O.}`
		` UNION {?I rdf:type C. O R ?I.}}`

3a
3b
3a+b
	Get all objects of type C in *any* relation with object O	`SELECT ?I WHERE {?I rdf:type C. ?I ?R O.}`
		`SELECT ?I WHERE {?I rdf:type C. O ?R ?I.}`
		`SELECT ?I WHERE {{?I rdf:type C. ?I ?R O.}`
		` UNION {?I rdf:type C. O ?R ?I.}}`

4a
4b
4a+b
	Get all objects of *any* type in a relation R with object O	`SELECT ?I WHERE {?I R O.}`
		`SELECT ?I WHERE {O R ?I.}`
		`SELECT ?I WHERE {{?I R O.} UNION {O R ?I.}}`

5a
5b
5a+b
	Get all objects of *any* type in *any* relation with object O	`SELECT ?I WHERE {?I ?R O.}`
		`SELECT ?I WHERE {O ?R ?I.}`
		`SELECT ?I WHERE {{?I ?R O.}`
		`UNION {O ?R ?I.}}`

Table 1. The different query types we used to analyze the performance impact of different variants, and their SPARQL representation.

4 Evaluation

4.1 Query Times

In this section, we analyze the performance impact of the different connector rule and caching variants on different reasoning tasks which involve information about and in the running system. We have examined both the *query times* for different query types, as well as the maximum degree of dynamics, i.e. the *maximum frequency of updates* to the A-box which can be handled by the different implementation variants.

4.2 Setup

To evaluate the impact of the different variants, we have analyzed various elementary query types, as shown in table 1. The queries were selected to cover a variety of cases with known or unknown predicate types, as well as known or unknown type restrictions for the queried objects. For each query, we have measured the average time to process that query. All tests have been carried out on a Windows XP 64Bit PC with an Intel Core Duo 3.00GHz processor and 4GB of RAM, using Java 1.6 and OntoBroker 5.3.

The variables in our evaluation are the number of software components (5, 10, and 20), the number of object instances maintained by each component (250 and 500, randomly distributed over 10 classes per component), and the average update frequency (25 and 50 updates per second). Each instance has been given 5 relations to other instances. Therefore, our maximum test set has 10000 object instances with 50000 relations. As our focus is on *dynamic* data, we altered the data at a frequency of 50 updates per second in the first set of evaluations.

In Fig. 3, we show the results of selected typical queries, which illustrate the main findings of our analysis[2]. While the figure depicts the results for the $a + b$ flavor of each query type (see table 1), the results for the a and b flavor are similar.

Generally, the query times using eager caching are faster than those using lazy caching. The actual factor between the two ranges from double speed (e.g., in case of type 4 queries) to only marginal improvements (e.g., in case of type 3 queries). Since lazy caches have to re-create the invalidated triples to answer a query, eager caches can always serve the requested triples directly. Therefore, the latter can answer queries faster.

When looking at scalability, it can be observed that doubling the number of integrated components (which also doubles the number of A-box instances) about doubles the query answer time in most cases, thus, there is a linear growth of complexity in most cases.

Multiple observations can be made when looking at the individual results regarding different queries. For type 3 queries, it is single rules which produce significant outliers (see Fig. 3(b)), while for type 5 queries, it is generic rules (see Fig. 3(d)). Thus, only only extended rules guarantee reasonable response times in all cases without any outliers, although they are outperformed by generic rules in type 2 and 4 queries.

In case of type 1, 2 and 4 queries, the relation in which the objects sought is already fixed (e.g., "find all persons which *are married to* another person"), while the case of type 3 and 5 queries, the relation is variable (e.g., "find all persons which have *any relation* to another person"). The first type of query is rather *target-oriented*, while the latter is rather *explorative*. The key finding of the results is that target-oriented queries do not pose any significant problems, while explorative queries do.

The bad behavior of single rules in the case of explorative queries can be explained by the fact that when an explorative query is answered, the various single rules fire, thus causing many potentially expensive invocations of the connector – for N components and M types, an explorative query may cause up to $O(N \times M)$ connector invocations. The generic and extended rules, on the other hand, invoke the connector less often. For a similar reason, the generic rule variant behaves badly for the explorative query types: here, the reasoner determines the possible relation types by invoking the wrapper multiple times, each time trying another relation type. Extended and single rules, on the other hand, already restrict the relation types in the rule body, thus requiring less invocations of the wrapper.

[2] Type 1 queries are not shown; they are generally answered very quickly, and there are no significant differences between the approaches.

4.3 Maximum Frequency of A-box Updates

Besides the time it takes to answer a query, another important performance criterion is the robustness of the system regarding A-box dynamics. While the rule design only influences the query times as such, a careful design of the wrappers' caches has a significant impact on the system's scalability with respect to the maximum possible frequency of A-box updates, as shown in Fig. 4.

The figure shows that while both eager and lazy caches do not drop in performance too strongly when scaling up the number of instances involved, lazy caching is drastically more robust regarding A-box dynamics. While several thousand updates per second on the A-box are possible with lazy caching, eager caching allows for less than 100. As assumed, lazy caches thus scale up much better regarding a-box dynamics, but at the trade-off of longer query response times, as shown above.

5 Examples

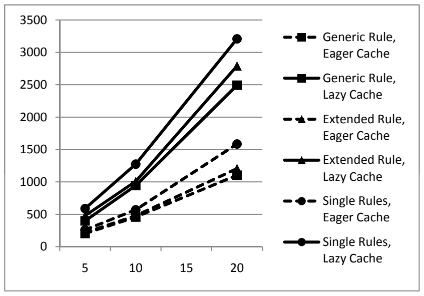
To illustrate the relevance of the findings presented in the previous sections, we introduce two examples: one using *goal-directed* and one using *explorative* queries.

5.1 Example for Goal-Directed Queries: Semantic Event Processing

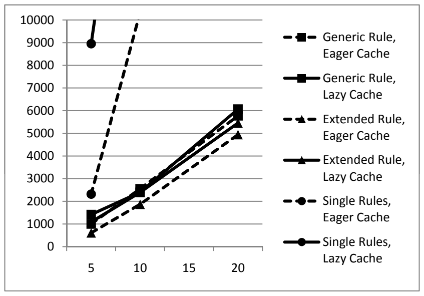
In [14], we have discussed the use of ontologies for application integration on the user interface level. The approach relies on using ontologies for formally describing user interface components and the information objects they process. Reasoning is used to facilitate semantic event processing as an indirection for decoupling the integrated applications [9].

By annotating the events produced by different user interface components, a reasoner can analyze those events, compute possible reactions by other components, and notify those components for triggering those reactions. This reasoning process requires instance information about the different applications, their states, and the data they process, which is delivered by the framework explained in Sect. 2.

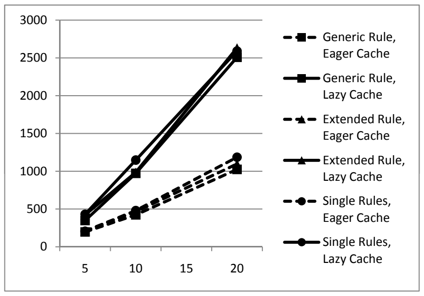
An example for an integration rule could state the following: "When the user performs a select action with an object representing a customer who has an address, the address book component will display that address, if it is visible on the screen." If this rule is evaluated by a reasoner, it has to be able to validate certain conditions, e.g. whether there is an address book component which is visible, or whether the customer in question has an address. It therefore needs access to information about both the system's components as well as the information objects they process. More sophisticated reasoning may come into place, e.g., when implementing different behaviors for standard and for premium customers, where the distinction between the two may involve the evaluation of different business rules.



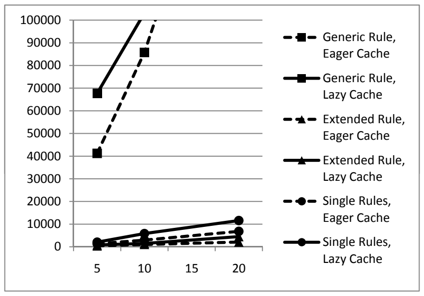
(a) Query type 2a+b



(b) Query type 3a+b



(c) Query type 4a+b



(d) Query type 5a+b

Fig. 3. Query times for selected query types, each for 500 object instances per component, and 50 updates per second. The x axis shows the number of components (there have been no explicit measurements for 15 components), and the y axis shows the query time in seconds.

A typical query used for event processing asks: given a particular event `#E`, which other events are triggered by that event:

```
SELECT ?E1 WHERE {?E1 #triggeredBy #E.}
```

Thus, the predicate is fixed, and the query is goal-directed. As discussed for the general results above, we have experienced that the different invocation rule variants do not affect performance too much, while eager caching leads to a significant speed-up. Details on the example can be found in [9].

5.2 Example for Explorative Queries: Semantic Data Visualization

Gathering and aggregating information from different IT systems can be a cumbersome and time consuming task for the end user. Combining that data with a reasoner can provide direct benefit for the end user.

In [8], we have introduced the *Semantic Data Explorer*. The Semantic Data Explorer provides a uniform graphical visualization of the data contained from applications using a central reasoning module, using the architecture discussed

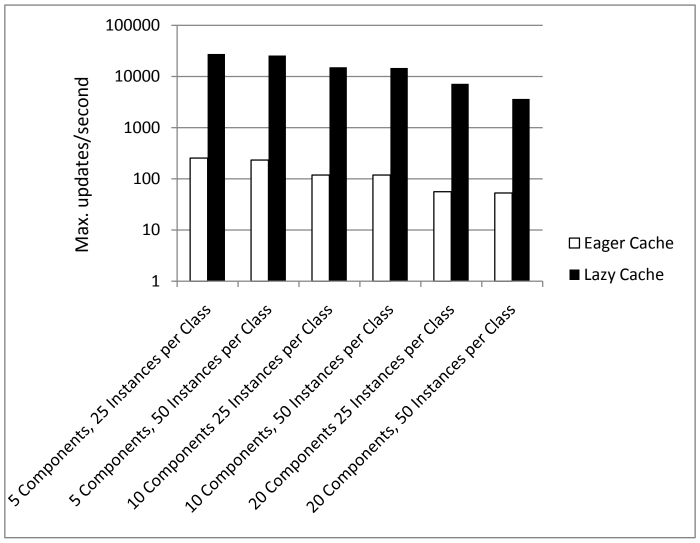


Fig. 4. Evaluation of robustness of different caching approaches regarding A-box dynamics. The graph shows the maximum number of updates per second that the system can process. Note that the y-axis has a logarithmic scale.

above. The user can drag objects from connected applications to the Semantic Data Explorer and navigate the corresponding graph view.

The Semantic Data Explorer uses the reasoner as an indirection for constructing the graph view. From an implementation point of view, this architecture provides a decoupling of the visualization and the data sources. More importantly, the reasoner may also reveal *implicit knowledge* gathered from the A-box information using T-box axioms and rules. This implicit knowledge is then included in the visualization as well, providing additional value to the end user.

A user study has shown that the Semantic Data Explorer can lead to significantly faster task completion times when gathering information, as well as to enhanced user satisfaction[3]. Implementation details on the tool, as well as a detailed description of the user study can be found in [8].

Displaying a node in the Semantic Data Explorer requires finding all incoming and outgoing edges to other objects and data properties. Thus, the underlying queries are *explorative*:

```
SELECT ?R ?V WHERE {<#x> ?R ?V}
SELECT ?R ?V WHERE {?V ?R <#x>}
```

The results above advise to use extended rules for this sort of queries. In fact, experiments with the SDE showed that using an extended rule leads to a perceivably faster system, which in turn increases the end users' satisfaction.

[3] A demo video is available at `http://soknos.de/index.php?id=470&L=0`.

6 Related Work

One of the best researched approaches for reasoning on objects from integrated systems is the use of so-called *wrappers* [11] or *mediators* [13], which collect objects from databases or structured documents and provide them to a reasoner as instance data.

D2RQ [15] is an example for a wrapper platform that integrates standard, non-RDF databases as RDF data sources and thus makes them available to a reasoner. Based on mapping rules, data entries from database tables are lifted as RDF instance data. The authors present an evaluation based on different query types that shows that retrieval of the data is feasible in reasonable time.

Lixto [16] is an example that uses wrappers to gather RDF data from non-annotated web pages. It provides graphical interfaces for defining the mechanisms used to extract data from the HTML documents. The authors show different use cases where RDF data gathered from the web is utilized. Those applications do not perform the retrieval at run-time, but offline, i.e. they parse web sites and build an RDF data store. The user's queries are then posed against that RDF data store.

OntoBroker [11] is a reasoning engine that provides different means for integrating data from various sources, including access to databases, web pages and web services via so-called *connectors*. As the API also foresees the integration of own connectors accessing arbitrary sources of instance data, we have based the prototype described in this paper on OntoBroker.

Various approaches have been proposed for directly accessing objects of running software applications [17]. There are two main variants of making the instance data known to the reasoner. The first relies on semantic annotation of the underlying class models, such as *sommer*[4] or *otm-j* [18]. The second uses class models generated directly from an ontology, with the necessary access classes for reasoning access being generated as well, such as *RDFReactor* [19], or *OntoJava* [20]. With dynamically typed scripting languages, the corresponding classes may also be generated on the fly, as shown, e.g., with *Tramp*[5] for Python. A detailed comparison of such approaches is given in [10]. However, analyses of efficiency and scalability of these approaches are rarely found.

Most of those approaches are not very flexible with respect to *conceptual heterogeneity* (i.e., class models that are different from the common ontology used for integration) as well as *technological heterogeneity* (i.e., using class models in different programming languages in parallel). The framework discussed in this paper uses flexible mapping rules and allows for containers for different programming languages [10, 21].

One of the best-known and most compelling application of making data from various applications known to a reasoner is the *semantic desktop* [3]. It allows users to browse, analyze, and relate data stored in different applications and provides new means of accessing data stored on a personal computer. Different

[4] `https://sommer.dev.java.net/`

[5] `http://www.aaronsw.com/2002/tramp/`

adapters exist which wrap data from databases, file systems, or e-mail clients. While there are various publications concerning impressive applications of the semantic desktop as well as various architectural aspects, systematic approaches of assessing the performance of the underyling technology are still hard to find.

7 Conclusion and Outlook

In this paper, we have introduced a framework for integrating dynamic A-box data from running software system with a central reasoner. There are several use cases for applying such a framework, e.g. searching information from different applications on a semantic desktop, dynamically adapting user interfaces to users' needs, or automatically integrating existing user interface components to a seamless application at run-time. In all of those approaches, a reasoner is used, which may need to have access to the data both contained *in* software components as well as *about* those software components as such. As reasoning is performed while those components are running, the A-box can be highly dynamic.

In most of the use cases of reasoning on dynamic systems sketched above, good performance is an essential requirement, as user interactions are involved. Based on the prototype implementation of our architecture, we have conducted several experiments to evaluate the performance impact of different implementation variants. Those variants encompass different caching strategies as well as the design of the rules from which the connectors are called. We have tested the variants with 13 different query types.

In this paper, we have analyzed the performance impact of three different rule types for rules invoking connectors to software components: generic rules, extended rules, and single rules. In some test cases, the query answering times even differ at a factor of 100 between the different approaches. This proves that the design of rules has a significant impact on the system performance.

One major finding is that there is no solution that provides optimal results for each usage scenario. In summary, we have shown that there are significant differences between *explorative* and *goal-directed* queries: in the first case, queries contain a fixed relation (e.g. "find all persons that *are married* to another person"), while in the latter case, the relation is a variable (e.g. "find all persons that *have any relation to* another person"). While some queries are handled almost equally well by all three rule types, only extended rules guarantee reasonable and stable query answering times in *all* cases. To illustrate the significance of the results, we have introduced two example use cases, one using goal-directed and one using explorative queries.

In addition, we have analyzed two different strategies for caching data in the wrappers: eager and lazy caching. Eager caching allows for response times up two twice as fast as lazy caching. On the other hand, lazy caching supports much more dynamic A-boxes: eager caching only works for less than 100 A-box updates per second, while with lazy caches, several thousand A-box updates per second can be processed. Therefore, a trade-off between A-box dynamics and query times can be identified. When implementing an actual system, a solution should be chosen according to that system's actual requirements.

In this paper, we have analyzed the performance effects using a set of elementary queries in this paper, and we have shown that different implementation variants perform better or worse with certain query types. More complex query types may reveal deeper insights into performance optimization of semantic applications.

While semantic technologies and reasoning on running software applications allow for interesting and valuable functionality, poor performance can be – and in fact often is – a show stopper. Thus, such applications should be carefully designed in order to be adopted by end users on a larger scale. With this paper, we have given insight in some strategies which can be carried over to the development of high performance systems using semantic technology. We are confident that this contribution will help developers of semantic web based software in creating systems which be come widely accepted.

Acknowledgements

The work presented in this paper has been partly funded by the German Federal Ministry of Education and Research under grant no. 01ISO7009 and 01IA08006.

References

1. Paulheim, H., Probst, F.: Ontology-Enhanced User Interfaces: A Survey. International Journal on Semantic Web and Information Systems **6**(2) (2010) 36–59
2. Cheyer, A., Park, J., Giuli, R.: IRIS: Integrate. Relate. Infer. Share. [22]
3. Sauermann, L., Bernardi, A., Dengel, A.: Overview and Outlook on the Semantic Desktop. [22]
4. Karim, S., Tjoa, A.M.: Towards the Use of Ontologies for Improving User Interaction for People with Special Needs. In Miesenberger, K., Klaus, J., Zagler, W.L., Karshmer, A.I., eds.: ICCHP. Volume 4061 of Lecture Notes in Computer Science., Springer (2006) 77–84
5. Gribova, V.: Automatic Generation of Context-Sensitive Help Using a User Interface Project. In Gladun, V.P., Markov, K.K., Voloshin, A.F., Ivanova, K.M., eds.: Proceedings of the 8th International Conference "Knowledge-Dialogue-Solution". Volume 2. (2007) 417–422
6. Kohlhase, A., Kohlhase, M.: Semantic Transparency in User Assistance Systems. In: Proceedings of the 27th annual ACM international conference on Design of Communication. Special Interest Group on Design of Communication (SIGDOC-09), Bloomingtion,, IN, United States, ACM Special Interest Group for Design of Communication, ACM Press (2009) 89–96
7. Paulheim, H.: Ontologies for User Interface Integration. In Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K., eds.: The Semantic Web - ISWC 2009. Volume 5823 of LNCS., Springer (2009) 973–981
8. Paulheim, H.: Improving the Usability of Integrated Applications by Using Visualizations of Linked Data. In: Proceedings of the International Conference on Web Intelligence, Mining and Semantics (WIMS'11). (2011)

9. Paulheim, H.: Efficient Semantic Event Processing: Lessons Learned in User Interface Integration. In Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T., eds.: The Semantic Web: Research and Applications (ESWC 2010), Part II. Volume 6089 of LNCS., Springer (2010) 60–74
10. Paulheim, H., Plendl, R., Probst, F., Oberle, D.: Mapping Pragmatic Class Models to Reference Ontologies. In: The 2011 IEEE 27th International Conference on Data Engineering Workshops - 2nd International Workshop on Data Engineering meets the Semantic Web (DESWeb). (2011) 200–205
11. Decker, S., Erdmann, M., Fensel, D., Studer, R.: Ontobroker: Ontology Based Access to Distributed and Semi-Structured Information. In Meersman, R., Tari, Z., Stevens, S.M., eds.: Database Semantics - Semantic Issues in Multimedia Systems, IFIP TC2/WG2.6 Eighth Working Conference on Database Semantics (DS-8), Rotorua, New Zealand, January 4-8, 1999. Volume 138 of IFIP Conference Proceedings., Kluwer (1999) 351–369
12. Angele, J., Lausen, G.: Ontologies in F-Logic. In Staab, S., Studer, R., eds.: Handbook on Ontologies. International Handbooks on Information Systems. 2nd edition edn. Springer (2009) 45–70
13. Wiederhold, G., Genesereth, M.: The Conceptual Basis for Mediation Services. IEEE Expert **12**(5) (sep/oct 1997) 38 –47
14. Paulheim, H., Probst, F.: Application Integration on the User Interface Level: an Ontology-Based Approach. Data & Knowledge Engineering Journal **69**(11) (2010) 1103–1116
15. Bizer, C., Seaborne, A.: D2RQ - Treating Non-RDF Databases as Virtual RDF Graphs. In: ISWC2004 Posters. (November 2004)
16. Baumgartner, R., Eiter, T., Gottlob, G., Herzog, M., Koch, C.: Information Extraction for the Semantic Web. In Eisinger, N., Maluszynski, J., eds.: Reasoning Web. Volume 3564 of Lecture Notes in Computer Science., Springer (2005) 275–289
17. Puleston, C., Parsia, B., Cunningham, J., Rector, A.: Integrating Object-Oriented and Ontological Representations: A Case Study in Java and OWL. In Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T.W., Thirunarayan, K., eds.: The Semantic Web - ISWC 2008. Volume 5318 of LNCS., Springer (2008) 130–145
18. Quasthoff, M., Meinel, C.: Semantic Web Admission Free - Obtaining RDF and OWL Data from Application Source Code. In Kendall, E.F., Pan, J.Z., Sabbouh, M., Stojanovic, L., Bontcheva, K., eds.: Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering (SWESE). (2008)
19. Völkel, M., Sure, Y.: RDFReactor - From Ontologies to Programmatic Data Access. In: Posters and Demos at International Semantic Web Conference (ISWC) 2005, Galway, Ireland. (2005)
20. Eberhart, A.: Automatic Generation of Java/SQL Based Inference Engines from RDF Schema and RuleML. In Horrocks, I., Hendler, J.A., eds.: The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings. Volume 2342 of Lecture Notes in Computer Science., Springer (2002) 102–116
21. Paulheim, H.: Seamlessly Integrated, but Loosely Coupled - Building UIs from Heterogeneous Components. In: ASE '10: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, ACM (2010) 123–126
22. Decker, S., Park, J., Quan, D., Sauermann, L., eds.: Proceedings of the ISWC 2005 Workshop on The Semantic Desktop - Next Generation Information Management & Collaboration Infrastructure. Volume 175 of CEUR-WS. (2005)