Delta-Reasoner: a Semantic Web Reasoner for an Intelligent Mobile Platform

Boris Motik Ian Horrocks
Department of Computer Science
Oxford University
Wolfson Building, Parks Road
OX1 3QD, Oxford, UK

Su Myeon Kim
Software Center, Samsung Electronics
416, Maetan-dong, Yeongtong-gu, Suwon-si
Gyeonggi-do
443-742 South Korea

ABSTRACT

To make mobile device applications more intelligent, one can combine the information obtained via device sensors with background knowledge in order to deduce the user's current context, and then use this context to adapt the application's behaviour to the user's needs. In this paper we describe Delta-Reasoner, a key component of the Intelligent Mobile Platform (IMP), which was designed to support context-aware applications running on mobile devices. Context-aware applications and the mobile platform impose unusual requirements on the reasoner, which we have met by incorporating advanced features such as incremental reasoning and continuous query evaluation into our reasoner. Although we have so far been able to conduct only a very preliminary performance evaluation, our results are very encouraging: our reasoner exhibits sub-second response time on ontologies whose size significantly exceeds the size of the ontologies used in the IMP.

Categories and Subject Descriptors

H.4.0 [**Information Systems**]: General

General Terms

Algorithms

Keywords

OWL, reasoning, context services

1. INTRODUCTION

The emergence of mobile devices with affordable data plans is reshaping the daily lives of millions of people around the world. As well as having access to all of the digital resources available via the Internet, such devices are additionally equipped with various sensors that allow a mobile device to detect and interact with its physical environment. For example, mobile devices nowadays routinely contain a GPS receiver, compass, accelerometer, and gyroscope, and they can typically connect to WiFi and Bluetooth networks. Using data obtained from these sensors, possibly in combination with information accessed via the Internet, a mobile device can identify the *context* that it is being used in. For example, the location of the device might be determined using GPS data, or by combining the SSIDs of the visible WiFi networks with (online) data about the location of WiFi networks. Furthermore, the visible Bluetooth devices might be used to identify nearby individuals. Finally, location and accelerometer data might be used to determine the user's current activity (such as running).

The goal of *context-aware* applications is to exploit the context information in an intelligent way for the purpose of helping the device owner in organising his daily life. For example, an application might remind the user to buy some milk when the context indicates that the user is located close to a grocery store. Similarly, an application might use the calendar information to remind the user of an upcoming family member's birthday. Since the practical advantages of such applications seem compelling, manufacturers are nowadays actively exploring ways of extending their products with context-awareness. However, although context-awareness has been the subject of much research, it has so far had relatively little impact on widely used applications. We believe that this is largely because existing work in context awareness has assumed an environment in which sensors and devices are linked to specific applications. Application developers and/or users are thus required to deal with each context-aware application individually, which increases implementation cost considerably, limits flexibility, and may make unrealistic demands on the user.

In order to overcome these limitations, we are developing a context-aware platform for mobile devices which we call the Intelligent Mobile Platform (IMP). The IMP can dynamically reconfigure itself in order to adapt to different situations and scenarios, and it provides context-aware services designed to support third-party applications. The goal of the IMP is to enable the development of a wide range of intelligent applications that can exploit the services provided in order to provide context-aware behaviour.

To support context-aware mobile applications, the IMP must manage information about the mobile device, its environment, the user, and the application scenario. Moreover, the requirements of context-aware applications are likely to exceed the simple storage and retrieval of information, so *semantic technologies* will therefore play a prominent role in the IMP's functionality. In particular, the IMP will use a *knowledge management system* and appropriate *reasoning services* to determine the current context of a mobile device and identify the appropriate actions. For example, based on the calendar information, the address book entries, the

WWW 2012, April 16–20, 2012, Lyon, France
ACM 978-1-4503-1230-1/12/04.

user's preferences, and the current location, a client application might use the IMP to recognise a context in which the user should be reminded to buy a birthday present.

To provide the required knowledge management and reasoning services, several nontrivial questions must be answered. The first question is how to represent the knowledge used for reasoning in an adequate way. While ad hoc mechanisms are certainly possible, they are unlikely to provide a sound foundation for the IMP. Instead, using a known and well-understood knowledge representation formalism offers numerous important advantages. Knowledge representation is a complex multifaceted problem for which no single solution exists; hence, depending on the requirements, different knowledge representation languages might be suitable for different applications. Furthermore, the choice of the knowledge representation language can impose theoretical limitations on the scalability of reasoning.

The second question is how to efficiently implement reasoning for the selected knowledge representation language. Various forms of reasoning can often be seen as (more or less) hard combinatorial problems. Exact reasoning algorithms are often of very high computational complexity (NP-complete or beyond), so effective heuristics are typically needed to facilitate reasoning with acceptable performance.

The third question concerns adequate tool support. Developing the necessary infrastructure from scratch can be very time consuming, so reusing existing software is likely to considerably reduce the development cycle.

In this paper we describe Delta-Reasoner—the knowledge management system used to support context-aware applications that was jointly developed by Samsung and Oxford University. In particular, we discuss how the above mentioned questions can be effective answered by basing the reasoner on the W3C standard languages: Delta-Reasoner uses the Resource Description Framework (RDF) [11] as the basic data model, and it uses the Web Ontology Language (OWL) [14] for representing the necessary background knowledge. RDF and OWL were developed by relying on the results of fundamental research into knowledge representation and automated reasoning [1].

Since the sensor readings change very frequently, Delta-Reasoner must continuously update the conclusions that it can draw from its ontology. Recomputing all conclusions from scratch every couple of seconds would clearly impose significant load on the limited resources of mobile devices. To this end, Delta-Reasoner uses *incremental* reasoning [18, 8]: after each change, our reasoner just computes the difference between the old and the new set of consequences, which considerably conserves resources.

The rest of this paper is structured as follows. In Section 2 we describe the context-aware setting and IMP in more detail. In Section 3 we present a brief overview of RDF, OWL, and reasoning systems. In Section 4 we discuss the application requirements that influenced the design of Delta-Reasoner. In Section 5 we present an outline of the reasoner's architecture. Finally, in Section 6 we present the results of a preliminary performance evaluation.

2. INTELLIGENT MOBILE PLATFORM

Figure 1 shows the typical environment for which the IMP is designed. In their daily lives, people usually move through a series of different situations, such as a home, a car, an office, or a restaurant. The main goal of the IMP is to determine the current situation based on various sensor readings. Sensors may include internal sensors (those embedded in the device, such as GPS), external sensors (those provided in a specific situation, such as an indoor location sensors), and even pseudo-sensors (virtual sensors such as weather information from Internet services). The list of available external sensors and pseudo-sensors may vary according to the situation, which will affect the services that the IMP can provide. Moreover, the kinds of sensor available and the kinds of services required may not be known at the point the IMP is being designed. Consequently, the IMP must be flexible enough to allow for extending the set of available sensors and/or services in a seamless way.

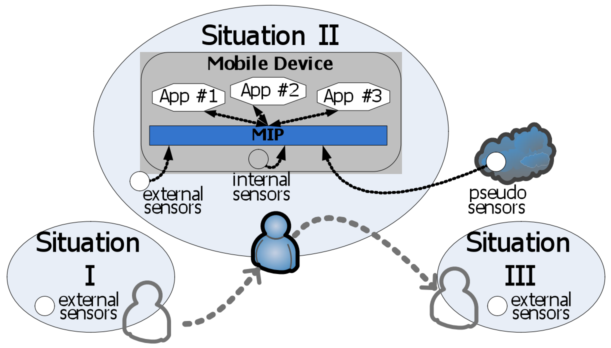

Figure 1: Typical Environment

The IMP exploits semantic technologies in order to provide the necessary flexibility and configurability. In particular, the IMP uses OWL ontologies to represent different situations and application scenarios, and it uses RDF to represent sensor and application data. The use of semantic technologies in context-aware engines has already been proposed, and our approach is related to that of Luther et al. [12]. While this work takes into account that the user's situation can change, the target application—service recommendation—is fixed at design time. In contrast, a change in the application setting can be reflected in the IMP by loading appropriate ontologies into the reasoner; thus, the system's behaviour can be changed without modifying the system itself. Our system thus relies on OWL reasoning to correctly interpret the given context and decide on appropriate application response. We refer to the set of ontologies available to a given application in a given context as the *context model*.

On initialisation, the context model contains only basic information, but it can be updated on the fly as the user's situation changes. Such updates may be *implicit* or *explicit*. Implicit updates occur, for example, when the user changes location, which may result in the IMP changing the model to reflect the new situation; such updates are typically valid for all applications. Explicit updates occur when an intelligent application wants to change its own behaviour; such updates are typically valid only for the application in question. In order to support explicit and implicit updates, the IMP must support the addition and retraction of information, and it must be able to reason w.r.t. a combination of context and application-dependent information.

For such an approach to be useful in practice, it is critical for the IMP to respond to the changes in the user's circumstances in a timely manner [21]. Achieving this goal is made difficult by the fact that the data acquired through the IMP's sensors can change very frequently, and that even the ontology representing the application scenario can change regularly. In the rest of this paper we present an outline of the technical solution to this problem.

Raw sensor data is typically too fine grained for direct use in context reasoning, and so it must be enriched via complex computations, such as filtering, pattern matching, and counting. For example, the user's current activity might be determined from accelerometer sensor data via activity prediction algorithms. As another example, the ambient temperature reading can be quantised into predetermined ranges. Such transformations can usually be implemented more efficiently in procedural code than by logical reasoning, so the IMP uses a sensor preprocessing layer (SPPL) between the sensors and the context reasoner. The SPPL provides certain common preprocessing functions, which can be easily combined to obtain complex preprocessing transformations; furthermore, SPPL can be easily extended with user-developed preprocessing functions. By quantising and interpreting sensor data, the SPPL can often reduce the number of updates to the context model, which considerably decreases the response times of the IMP.

3. STATE OF THE ART IN OWL AND RDF

Before proceeding with a discussion of how semantic technologies can be used to support context reasoning in practice, in this section we first present a brief overview of RDF, OWL, and the existing reasoning systems.

RDF is a metadata standard that allows users to make simple assertions. The elementary unit of information in RDF is a *triple*—an assertion of the form $\langle s, p, o \rangle$ stating an that object o is the value of a property p for a subject s. For example, that a store identified as $:s1$ is a Tesco store located at geographic longitude XYZ can be stated using the following triples:

$$\langle :s_1, rdf{:}type, :TescoStore \rangle \tag{1}$$

$$\langle :s_1, :hasLongitude, XYZ \rangle \tag{2}$$

Sets of triples are commonly called (RDF) *graphs* or *ABoxes*. While RDF is mainly used for making elementary assertions, OWL can be used to state axioms describing the structure of the application domain. In this paper we use the OWL 2 DL version of OWL—a version of the language that was designed to be decidable and thus allow for sound and complete practical reasoning. As an example, the following OWL 2 DL axiom states that all Tesco stores are grocery stores:

$$\mathsf{SubClassOf}(\ :TescoStore\ \ :GroceryStore\ ) \tag{3}$$

A set of axioms describing a domain of interest is commonly called a *TBox*. From triples (1)–(2) and axiom (3), one can conclude that 's_1 is a grocery store'—that is, one can derive triple $\langle :s_1, rdf{:}type, :GroceryStore \rangle$. Since reasoning with OWL 2 DL is of high computational complexity [1], three profiles of OWL 2 DL were developed: OWL 2 RL, OWL 2 QL, and OWL 2 EL [13]. These profiles can be understood as subsets of the OWL 2 DL language that offer favourable computational properties.

In order to extend its expressivity, the Semantic Web Rule Language (SWRL) [10] extension of OWL 2 DL was developed. SWRL allows users to state rules that check for complex conditions. For example, the following rule states that if the device location is the same as the location of another object, then the device is located near this object:

$$\begin{aligned} :deviceLocation(?P, ?L) \wedge :hasLocation(?O, ?L) \rightarrow \\ :isNear(?P, ?O) \end{aligned} \tag{4}$$

SPARQL [17] provides a standard language for querying Semantic Web systems. For example, the following query retrieves pairs of objects that are near to each other:

$$\mathsf{SELECT}\ ?X\ ?Y\ \mathsf{WHERE}\ \{\ ?X\ :isNear\ ?Y\ \} \tag{5}$$

Existing OWL reasoners can be broadly divided into two groups, according to the kind of implemented reasoning algorithm. In the first group are reasoners such as Pellet [16], HermiT [15], FaCT++ [19], and RACER [9] that are based on provably correct (hyper)tableau calculi. These calculi can typically handle all or most of OWL 2 DL, and they can solve a wide range of reasoning tasks, such as classifying an ontology and checking ontology entailment; hence, (hyper)tableau calculi are particularly useful in applications that require extensive reasoning about the schema (i.e., reasoning about the definitions of classes and properties). It is possible to implement conjunctive query answering (i.e., reasoning about the data) using hypertableau calculi [5]; however, such implementations are unlikely to be scalable (i.e., they are unlikely to be able to handle ontologies with large numbers of assertions).

In the second group are reasoners such as Jena [3], Jena2 [23], Oracle [4], and Sesame [2] that are based on (deductive) RDF stores. These systems typically store their data as RDF graphs, and they implement reasoning via *materialisation*: all (relevant) consequences that follow from RDF data and the OWL ontology are precomputed and stored in a preprocessing step; after that, user queries can be simply evaluated in the precomputed data. The main benefit of such an approach is that computation-intensive processing is concentrated in the preprocessing phase. Thus, the cost of preprocessing is amortised over multiple query evaluations, which makes this style of reasoning particularly attractive for data-intensive applications. Consequently, most reasoners in this group aim at providing high levels of scalability—that is, they were designed to support the management of RDF graphs containing hundreds of millions or even billions of triples. Materialisation, however, has two main drawbacks. First, materialisation can significantly increase the size of the RDF data set, thus requiring efficient storage and management of even larger materialised data sets. Second, materialisation can be applied only to OWL 2 RL ontologies. Thus, all reasoners in this group support only OWL 2 RL or a fragment thereof, with some also supporting (a fragment of) SWRL.

4. REQUIREMENTS

As we explain in this section, the requirements placed on a reasoner supporting context reasoning are quite distinct from the requirements commonly found in applications of Semantic Web technologies. Consequently, off-the-shelf reasoners are unlikely to provide a good knowledge management platform for context reasoning applications. Based on this discussion, in Section 5 we present the architecture of the reasoner powering the IMP.

4.1 Focus on ABox Reasoning

The context of a mobile device can be determined at any given point in time by checking whether the sensor data satisfies certain conditions. Such an approach can be realised using Semantic Web technologies in the following way:

- One can develop an ontology of terms for describing sensor values. This ontology might contain properties such as :*deviceLocation*.

- The current values obtained from the sensors can be encoded by an ABox over the vocabulary of the sensor ontology. For example, the current location of the mobile device might be represented by instantiating the :*deviceLocation* property.

- The conditions for recognising the current context can be encoded using ontology axioms. These may involve a mix of TBox axioms, ABox assertions, and SWRL rules. For example, using axioms such as (3) one can develop a vocabulary for describing real-world objects such as shops and other points of interest. Furthermore, using triples such as (1) and (2) one can describe individual points of interest. Finally, using rules such as (4) one can formalise the conditions for interpreting the sensor data and detecting the current context.

- At run-time, the current context can be determined by evaluating a query over the current ABox and the ontology.

The primary inference in such an application is thus answering queries over a data set. Consequently, reasoners based on (hyper)tableau techniques may not be the first choice for the IMP, as these reasoners are designed primarily for schema reasoning, rather than query answering.

4.2 Small Data Sets

Systems such as Jena, Oracle, and Sesame were designed to handle ABoxes with millions or even billions of triples. These systems, however, are typically not constrained by the available processing resources: today, even relatively inexpensive servers offer large amounts of RAM and disk space, as well as excellent processing performance.

A reasoner supporting a context reasoning application, however, has to satisfy very different requirements. ABoxes in such an application are unlikely to be very large: we estimate that they will contain at most about 100,000 triples. In contrast, the processing resources available to the reasoner are severely constrained: the target hardware platform typically offers a modest processor and limited RAM, which must be shared between several processes. Consequently, a reasoner supporting a context reasoning application should follow a lightweight approach to design that allows the reasoner to efficiently exploit the available resources.

4.3 Incremental Reasoning and Querying

In traditional applications of Semantic Web technologies, the data typically changes slowly; for example, in many applications the data changes only daily or even weekly. Therefore, even if materialising all consequences of a data set takes a lot of time, this problem can usually be overcome by using a sufficiently powerful platform for offline preprocessing of the data set. Furthermore, in conventional Semantic Web applications, data updates mainly involve additions rather than retractions.

In contrast, the data in a context reasoning applications is likely to change very frequently. The sensors will produce fresh readings several times per second, and each reading is likely to be different from the previous one. Although the impact of such changes is mitigated by the sensor preprocessing layer, the ABox is still likely to change frequently, so recomputing the materialisation from scratch after every update seems unacceptable. Consequently, to achieve the desired levels of performance, the reasoner must be able to efficiently handle ABox updates, consisting of both additions and retractions; in other words, the reasoner must support *incremental reasoning*.

As explained in Section 2, an important service provided by the IMP is to track the device context and notify client applications when the context changes. Such functionality can be effectively supported via *continuous querying*—that is, by requesting the reasoner to notify the relevant clients whenever the answer to a query of interest changes. As we discuss in Section 5, such functionality can be implemented using incremental reasoning techniques.

4.4 Navigating the Class Hierarchy

SPARQL cannot express a query that retrieves only the *direct* superclasses of a class. This is primarily because Semantic Web languages are based on first-order logic, which is *monotonic*: one cannot invalidate a conclusion by adding triples and/or axioms. A query for direct superclasses of a class, however, is clearly *nonmonotonic*, and so it cannot be expressed in OWL and SPARQL.

Determining direct superclasses of a class, however, is quite important for context reasoning. For example, one might describe the types of contexts as follows:

$$\mathsf{SubClassOf}(\,{:}\mathit{UniversityContext}\ \ {:}\mathit{WorkContext}\,) \tag{6}$$

$$\mathsf{SubClassOf}(\,{:}\mathit{WorkContext}\ \ {:}\mathit{Context}\,) \tag{7}$$

By the semantics of OWL, if the current context gets classified as an instance of :*UniversityContext*, the current context will also get classified as an instance of :*WorkContext* and :*Context* as well. In practice, however, one needs to distinguish the first from the latter two classes, as one is typically only interested in the most specific context. Thus, while queries about the class hierarchy are also important in other applications of Semantic Web technologies, they seem particularly important for context reasoning.

Problems such as the one mentioned above are often solved in practice by issuing several SPARQL queries and combining their results manually. Such solutions, however, are typically hard-coded to handle only specific types of problems, and thus lack flexibility. For example, combining class hierarchy queries with instance queries then becomes much more involved, both from the query specification and the query evaluation points of view. Thus, the lack of adequate support for hierarchy navigation seems to be an important feature that has not been adequately treated in the existing Semantic Web standards.

5. REASONER'S ARCHITECTURE

As one can see from our discussion in Section 4, a reasoner for the IMP should satisfy quite distinct requirements from those commonly found in Semantic Web applications. We

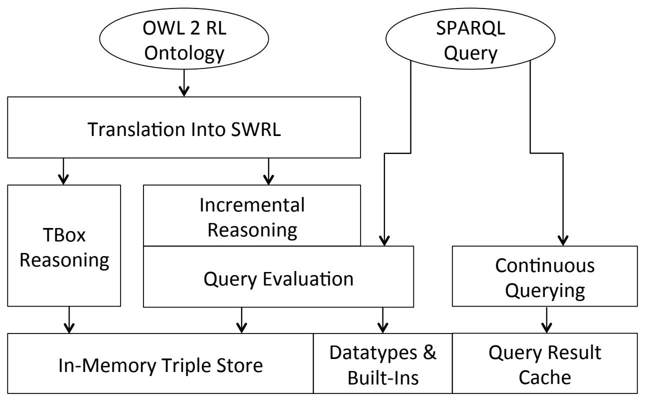

Figure 2: Reasoner's Architecture

therefore developed a completely new system called *Delta-Reasoner*, the architecture of which is shown schematically in Figure 2. In the following sections we discuss in more detail various components of the reasoner.

5.1 Translation into SWRL

As explained in Section 4.1, the main reasoning task in context reasoning is answering queries over moderately sized ABoxes. Consequently, we selected the OWL 2 RL profile of OWL as the target reasoning language. There are two main benefits of using OWL 2 RL. First, the language can readily be extended with SWRL rules without making the resulting formalism undecidable. This is particularly important because most conditions for recognising the current context can be easily expressed as SWRL rules. Second, reasoning with OWL 2 RL ontologies can be implemented in polynomial time using techniques known from deductive databases, which is particularly attractive given the constraints on the hardware resources of mobile devices.

Reasoning with OWL 2 RL ontologies is typically implemented by the following two-step process.

- The input ontology is loaded into an RDF triple store.
- The fixed set of rules given in [13, Section 4.3] is exhaustively applied to the RDF triples to derive certain triples that follow from the ontology. These rules can be divided into two parts: the rules in [13, Tables 4–8] derive ABox consequences, and the rules in [13, Table 9] derive TBox consequences. Rule (8) is an example of the former, while rule (9) is an example of the latter.

$$\begin{aligned}&\langle ?X, \mathit{rdf}{:}\mathit{type}, ?C_1\rangle \wedge \\ &\langle ?C_1, \mathit{rdfs}{:}\mathit{subClassOf}, ?C_2\rangle \rightarrow \\ &\qquad\qquad \langle ?X, \mathit{rdf}{:}\mathit{type}, ?C_2\rangle\end{aligned} \tag{8}$$

$$\begin{aligned}&\langle ?C_1, \mathit{rdfs}{:}\mathit{subClassOf}, ?C_2\rangle \wedge \\ &\langle ?C_2, \mathit{rdfs}{:}\mathit{subClassOf}, ?C_3\rangle \rightarrow \\ &\qquad\qquad \langle ?C_1, \mathit{rdfs}{:}\mathit{subClassOf}, ?C_3\rangle\end{aligned} \tag{9}$$

There are, however, several problems with such an approach to implementing OWL 2 RL. First, certain rules for deriving ABox consequences are quite complex and can therefore be difficult to evaluate. For example, rule (8) contains two atoms in the body, so to evaluate the rule one must compute a join. Second, the rules for deriving TBox consequences do not provide any completeness guarantees. For example, consider the OWL 2 RL ontology consisting of the following three axioms:

$$\mathsf{SubClassOf}(\ A\ \ B\ ) \tag{10}$$

$$\mathsf{SubClassOf}(\ A\ \ C\ ) \tag{11}$$

$$\mathsf{SubClassOf}(\ \mathsf{ObjectIntersectionOf}(\ B\ \ C\ )\ \ D\ ) \tag{12}$$

These axioms imply the following subsumption relationship between classes:

$$\mathsf{SubClassOf}(\ A\ \ D\ ) \tag{13}$$

The rules from [13, Table 9], however, fail to derive this consequence: axioms (10) and (11) imply that class A is subsumed by the intersection of classes B and C, but no rule from [13, Table 9] can draw this conclusion. One might try to extend the rules from [13, Table 9]. We, however, do not know of a fixed set of rules that is guaranteed to correctly compute all TBox consequences for all OWL 2 RL ontologies; in fact, developing such a set of rules seems to us like a very challenging task.

Because of these drawbacks, Delta-Reasoner employs a different implementation approach. Given an OWL 2 RL ontology, our reasoner loads only the triples corresponding to the ABox into its internal triple store. Furthermore, instead of using a fixed set of rules for reasoning, Delta-Reasoner translates the TBox of the ontology into a set of rules that is *specific to that ontology.* For example, instead of using a generic rule such as (8) to capture the semantics of the class hierarchy, Delta-Reasoner introduces a rule of the form (15) for each axiom of the form (14).

$$\mathsf{SubClassOf}(\ C_1\ \ C_2\ ) \tag{14}$$

$$\langle ?X, \mathit{rdf}{:}\mathit{type}, C_1\rangle \rightarrow \langle ?X, \mathit{rdf}{:}\mathit{type}, C_2\rangle \tag{15}$$

Rules such as (15) contain only one atom in the body and are thus easier to evaluate since they do not require the computation of joins. The translation is implemented as a straightforward extension of the translation described in [6]. The resulting rules correctly capture all ABox consequences that the reasoner needs to draw.

5.2 TBox Reasoning

As explained in the previous section, implementing complete TBox reasoning by a fixed set of rules seems like a challenging task. Consequently, it seems more sensible to derive TBox consequences using a sound and complete reasoner and then simply write the obtained consequences into the in-memory triple store. For example, one can use a reasoner such as HermiT to classify the knowledge base and then simply convert the resulting class hierarchy into triples.

Delta-Reasoner uses a variation of the approach outlined above. More concretely, Delta-Reasoner analyses the rules obtained from the input ontology to extract the information about subclasses, domains, and ranges. It then computes the class hierarchy using a proprietary algorithm. For practical reasons Delta-Reasoner does not use a reasoner such as HermiT for ontology classification; however, a sound and complete TBox reasoner could be used in principle. Finally, the class hierarchy and the domain/range information is written into the internal triple store. In the latter step, direct class subsumptions are encoded using a proprietary *rdfs:directSubClassOf* property. This allows the users to formulate queries about direct class subsumptions and thus satisfies the requirement described in Section 4.4.

5.3 Triple Store

The in-memory triple store is core a component of Delta-Reasoner that is responsible for storing ABox triples. From a logical point of the view, the triple store can be understood as simply a set of facts of the form

$$P(t_1, \ldots, t_n) \tag{16}$$

where P is called a *predicate*, and each t_i is either an individual or a literal. In addition to storage, the triple store is responsible for triple retrieval. A *retrieval query* Q can be understood as a fact of the form (16) in which each t_i is an individual, a literal, or a special symbol $*$. The result to such Q contains each fact A in the triple store that differs from Q only in the occurrences of $*$. For organisational reasons, all assertions in the triple store that share the same predicate are grouped together into *extensional tables.* Thus, the triple store can equivalently be viewed as a collection of extension tables.

ABox triples are all stored in the triple store as facts of the following form:

$$\mathsf{T}(s, p, o) \tag{17}$$

Thus, the extension table for the T predicate is the main component of the triple store, and it was realised by adapting the approach adopted in Hexastore [22]. In order to facilitate efficient retrieval, Delta-Reasoner employs three hierarchical indexes: I_{spo}, I_{pos}, and I_{osp}. Index I_{spo} is schematically shown in Figure 3. The index consists of three hash tables: the hash table at level 1 provides an index over the s-component of triples; the hash table at level 2 provides an index over the *sp*-components; and the hash table at level 3 provides an index over the *spo*-components. In addition, all entries in the level 1 hash table are arranged in a linked list; similarly, all entries in the level 2 hash table that share the same s-component are arranged in a linked list; finally, all entries in the level 3 hash table that share the same *sp*-components are arranged in a linked list. In this way, I_{spo} can be used to answer retrieval queries of the form $\mathsf{T}(s,p,o)$, $\mathsf{T}(s,p,*)$, $\mathsf{T}(s,*,*)$, and $\mathsf{T}(*,*,*)$. Index I_{pos} is defined analogously, and it can be used to answer retrieval queries of the form $\mathsf{T}(*,p,o)$ and $\mathsf{T}(*,p,*)$; finally, index I_{osp} can be used to answer retrieval queries of the form $\mathsf{T}(*,*,o)$ and $\mathsf{T}(s,*,o)$. These three indexes thus cover all possible retrieval queries, which allows for efficient retrieval of facts stored in the triple store.

As explained in the following sections, Delta-Reasoner often needs to distinguish triples that were explicitly asserted by the user from the triples that were inferred using a set of SWRL rules. For example, a user can delete only the explicitly asserted triples, and the implicitly derived triples are then modified automatically as needed to ensure consistency. To this end, the extension table for the T predicate is internally split into extension tables for predicates T_e and T_i, which contain the explicit and the implicit triples, respectively. The extension tables for T_e and T_i are implemented as 'views' over the extension table for the T predicate: each triple in the triple store is associated with a flag that determines whether the triple belongs to T_e or to T_i.

In addition to the extension table for the T predicate, the triple store also maintains extension tables for several auxiliary predicates that are used during reasoning. All these extension tables are implemented analogously to the extension table for the T.

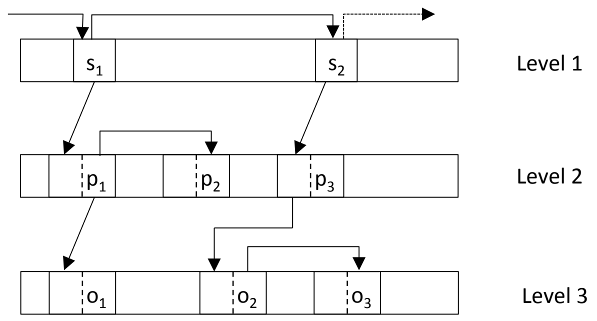

Figure 3: Index I_{spo}

5.4 Datatypes and Built-Ins

Delta-Reasoner can be easily extended with new datatypes and built-in predicates. To extend the reasoner with a new datatype, one mainly needs to provide a procedure for parsing the lexical form of the literals of the new datatype, and associate this procedure with the datatype URI.

Built-in predicates are realised as 'virtual' extension tables. For example, consider the comparison built-in predicate *swrlb*:*lessThan*: the predicate takes two arguments, and a fact of the form

$$\mathit{swrlb}{:}\mathit{lessThan}(t_1, t_2) \tag{18}$$

is true if the value of t_1 is smaller than the value of t_2. Clearly, there are infinitely many facts of the form (18), so listing all facts explicitly is infeasible. Therefore, the built-in is implemented as an extension table capable of answering only retrieval queries in which all arguments are different from $*$; in other words, this extension table requires all arguments of a retrieval query to be *bound.* As another example, consider the built-in *swrlb*:*subtract*: the built-in takes three arguments, and a fact of the form two arguments, and a fact of the form

$$\mathit{swrlb}{:}\mathit{subtract}(t_1, t_2, t_3) \tag{19}$$

is true if the value if t_1 is equal to t_2 minus t_3. Since materialising all facts of that form is infeasible, the built-in is implemented as an extension table capable of answering retrieval queries of the form (19) or $\mathit{swrlb}{:}\mathit{subtract}(*, t_2, t_3)$. In both cases, the built-in expects the values for t_2 and t_3 to be provided in the retrieval query. If t_1 is provided as well, then the implementation checks whether $t_1 = t_2 - t_3$; otherwise, the implementation returns the fact of the form (19) in which t_1 is replaced with $t_2 - t_3$.

5.5 Query Evaluation

Conjunctive queries are the basic type of query that Delta-Reasoner can evaluate. From a logical point of view, a conjunctive query can be understood as a rule of the following form, where $?\vec{x}$ stands for a vector of variables and each B_i is of the form (16) in which each t_j is allowed to be a variable:

$$Q(?\vec{x}) \leftarrow B_1 \wedge \ldots \wedge B_n \tag{20}$$

Note that by using B_i of the form $\mathsf{T}(s_i, p_i, o_i)$ and of the form (16) with P a built-in predicate allows conjunctive queries to capture SELECT–WHERE–FILTER queries of SPARQL.

Delta-Reasoner evaluates conjunctive queries using index nested loop joins. More precisely, it identifies all facts that match B_1; for each match, the reasoner identifies facts that match B_2; and so on. Such an algorithm is appropriate since the triple store is capable of answering all possible retrieval queries efficiently. Furthermore, the amount of space that this algorithm uses depends only on the size of the query and not on the number of the triples in the triple store, which makes the algorithm particularly suitable for usage on platforms with limited memory.

It is well known that the performance of such an algorithm can significantly depend on the order in which the atoms are evaluated. Determining the optimal order, however, is a hard combinatorial problem, whose solution depends on the existence of statistics about the data in the reasoner. The management of statistics information, however, is likely to incur significant overhead, particularly in situations when data changes frequently. Furthermore, since the reasoner is expected to manage a moderate amount of data, selecting the best order is unlikely to be critical. Consequently, the reasoner uses a very simple greedy reordering strategy whose goal is to propagate bindings as early as possible, while satisfying the restrictions on the retrieval queries for the built-in predicates occurring in the query.

5.6 Incremental Reasoning

Delta-Reasoner implements the incremental reasoning algorithm presented in [8, Section 7]. The algorithm starts with a set of rules R of the form (21), a set of *explicit* facts F_e of the form (16), and a set of *implicit* facts F_i obtained by applying all the rules in R to F_e.

$$H \leftarrow B_1 \wedge \ldots \wedge B_n \tag{21}$$

When given a set of facts F_e^- that are to be deleted from F_e, and a set of facts F_e^+ that are to be added to F_e, algorithm transforms F_e and F_i into sets of facts F_e' and F_i' such that

- $F_e' = (F_e \setminus F_e^-) \cup F_e^+$ and
- F_i' contains precisely all the facts obtained by applying all the rules in R to F_e'.

Clearly, this goal could be achieved by simply computing F_e' as outlined above and then computing F_i' from scratch (i.e., without taking F_i into account). If, however, F_e^- and F_e^+ affect only a small portion of F_e, this will involve recomputing most of F_i. The algorithm from [8] addresses this issue by computing F_i' incrementally, repeating as little work as possible. A complete presentation of this algorithm is beyond the scope of this paper; instead, we simply illustrate how the algorithm works when applied to a set R containing rules (22)–(23), and a set F_e containing facts (24)–(25).

$$\mathsf{T}(?x, \textit{rdf}:\textit{type}, A) \leftarrow \mathsf{T}(?x, \textit{rdf}:\textit{type}, B) \tag{22}$$
$$\mathsf{T}(?x, \textit{rdf}:\textit{type}, A) \leftarrow \mathsf{T}(?x, \textit{rdf}:\textit{type}, C) \tag{23}$$
$$\mathsf{T}_e(s, \textit{rdf}:\textit{type}, B) \tag{24}$$
$$\mathsf{T}_e(s, \textit{rdf}:\textit{type}, C) \tag{25}$$

Thus, the set F_i initially contains $\mathsf{T}_i(s, \textit{rdf}:\textit{type}, A)$. Please remember that T_e and T_i are just 'views' over T.

In order to store the sets of facts F_e^- and F_e^+, Delta-Reasoner uses auxiliary extension tables for predicates T_e^- and T_e^+. Furthermore, Delta-Reasoner also uses auxiliary extension tables for predicates T_i^- and T_i^+, which will contain implicit triples that should be respectively deleted from and inserted into F_i. For the purposes of our example, let us assume that we want to delete $\mathsf{T}(s, \textit{rdf}:\textit{type}, B)$ from and add $\mathsf{T}(t, \textit{rdf}:\textit{type}, B)$ to F_e; thus, F_e^- is initialised with the fact $\mathsf{T}_e^-(s, \textit{rdf}:\textit{type}, B)$, and F_e^+ is initialised with the fact $\mathsf{T}_e^+(t, \textit{rdf}:\textit{type}, B)$.

The incremental reasoning algorithm updates F_e and F_i using the following steps.

First, the set of rules R is transformed into a set of rules R_{del}, which is then evaluated over F_e, F_i, and F_e^- to compute the set of triples F_i^- that should be deleted from F_i. In our running example, R_{del} contains the following rules:

$$\mathsf{T}_i^-(?x, \textit{rdf}:\textit{type}, A) \leftarrow \mathsf{T}_e^-(?x, \textit{rdf}:\textit{type}, B) \tag{26}$$
$$\mathsf{T}_i^-(?x, \textit{rdf}:\textit{type}, A) \leftarrow \mathsf{T}_i^-(?x, \textit{rdf}:\textit{type}, B) \tag{27}$$
$$\mathsf{T}_i^-(?x, \textit{rdf}:\textit{type}, A) \leftarrow \mathsf{T}_e^-(?x, \textit{rdf}:\textit{type}, C) \tag{28}$$
$$\mathsf{T}_i^-(?x, \textit{rdf}:\textit{type}, A) \leftarrow \mathsf{T}_i^-(?x, \textit{rdf}:\textit{type}, C) \tag{29}$$

Roughly speaking, the rules in R_{del} are obtained by restricting the rules in R to the facts in $F_e^- \cup F_i^-$. In this way, the evaluation of R_{del} computes the consequences of R and the facts being deleted. In our running example, these rules ensure that F_i^- contains the fact $\mathsf{T}_i^-(s, \textit{rdf}:\textit{type}, A)$.

Second, all facts in F_e^- are removed from F_e, and all facts in F_i^- are removed from F_i. In our running example, $\mathsf{T}_i^-(s, \textit{rdf}:\textit{type}, A)$ is contained in F_i^-, so $\mathsf{T}_i(s, \textit{rdf}:\textit{type}, A)$ is removed from F_i. Note, however, that this deletion is 'too eager': $\mathsf{T}_i(s, \textit{rdf}:\textit{type}, A)$ is derivable from the explicitly asserted fact $\mathsf{T}_e(s, \textit{rdf}:\textit{type}, C)$ and rule (23).

Third, the set of rules R is transformed into a set of rules R_{red}, which is then evaluated over F_e, F_i, and F_i^- to 'rederive' the facts that were deleted in the previous step but that can still be derived from F_e. In our running example, R_{red} contains the following rules:

$$\mathsf{T}_i^+(?x, \textit{rdf}:\textit{type}, A) \leftarrow \mathsf{T}_i^-(?x, \textit{rdf}:\textit{type}, A) \wedge \mathsf{T}(?x, \textit{rdf}:\textit{type}, B) \tag{30}$$
$$\mathsf{T}_i^+(?x, \textit{rdf}:\textit{type}, A) \leftarrow \mathsf{T}_i^-(?x, \textit{rdf}:\textit{type}, A) \wedge \mathsf{T}(?x, \textit{rdf}:\textit{type}, C) \tag{31}$$

Roughly speaking, the rules in R_{red} are obtained by restricting the rules in R so that they check the 'rederivation' only of the facts in F_i^-—that is, of the facts deleted in the first two steps. In our running example, the evaluation of R_{red} over F_e, F_i, and F_i^- produces the fact $\mathsf{T}_i^+(s, \textit{rdf}:\textit{type}, A)$; this fact was 'eagerly deleted' in the previous two steps, and it will be later added back to F_i.

Fourth, the set of rules R is transformed into a set of rules R_{ins}, which is then evaluated over F_e, F_e^+, F_i, and F_i^+ to compute the facts that follow from the newly inserted facts. In our running example, R_{ins} contains the following rules:

$$\mathsf{T}_i^+(?x, \textit{rdf}:\textit{type}, A) \leftarrow \mathsf{T}_e^+(?x, \textit{rdf}:\textit{type}, B) \tag{32}$$
$$\mathsf{T}_i^+(?x, \textit{rdf}:\textit{type}, A) \leftarrow \mathsf{T}_i^+(?x, \textit{rdf}:\textit{type}, B) \tag{33}$$
$$\mathsf{T}_i^+(?x, \textit{rdf}:\textit{type}, A) \leftarrow \mathsf{T}_e^+(?x, \textit{rdf}:\textit{type}, C) \tag{34}$$
$$\mathsf{T}_i^+(?x, \textit{rdf}:\textit{type}, A) \leftarrow \mathsf{T}_i^+(?x, \textit{rdf}:\textit{type}, C) \tag{35}$$

Roughly speaking, the rules in R_{ins} are obtained by restricting the rules in R to the facts in $F_e \cup F_e^+ \cup F_i \cup F_i^+$. In this way, the evaluation of R_{ins} computes the consequences of R and F_e^+. In our running example, these rules derive $\mathsf{T}_i^+(t, \mathit{rdf}{:}\mathit{type}, A)$.

Fifth, all facts in F_e^+ are added to F_e, and all facts in F_i^+ are added to F_i. In our running example, $\mathsf{T}_e(t, \mathit{rdf}{:}\mathit{type}, B)$ is added to F_e, and $\mathsf{T}_i(s, \mathit{rdf}{:}\mathit{type}, A)$ and $\mathsf{T}_i(t, \mathit{rdf}{:}\mathit{type}, A)$ are added to F_i. At this point, F_e has been updated to reflect the changes in F_e^- and F_e^+, and F_i has been updated to correctly reflect all the consequences of R and F_e, so the algorithm terminates.

In all these steps, the rules are evaluated as queries whose answers are simply added as facts to the triple store. Furthermore, we modified this basic algorithm as explained in [20] to handle schema updates as well.

This algorithm seems appropriate to the context reasoning setting for several reasons. First, the modification of the rules ensures that the rules apply only to the facts being changed; thus, if F_e^- and F_e^+ are small, the evaluation of the rules should be reasonably efficient. Second, the algorithm does not require complex data structures for the management of dependencies between facts: it can be implemented by simply evaluating suitably modified sets of rules, and thus does not impose further requirements on the limited resources of a mobile device.

5.7 Continuous Querying

Delta-Reasoner allows clients to register a query of interest and then get a notification whenever the query's result changes. This functionality can straightforwardly be supported using the incremental reasoning algorithm outlined in Section 5.6. As explained in Section 5.5, a conjunctive query can be seen as a rule. Thus, the rules obtained from registered queries can simply be merged with the rules obtained from the input OWL 2 RL ontology; the incremental reasoning algorithm can then be used to maintain the materialisation of the query answers and for detecting when a query answer should change.

6. PERFORMANCE EVALUATION

In this section we present the results of a preliminary performance evaluation of the Delta-Reasoner.

Integration of the Delta-Reasoner into the IMP and the development of the appropriate ontologies are still ongoing. Consequently, we have so far been able to conduct only a preliminary performance evaluation using a standard laptop (a MacBook Pro with 4 GB RAM and an Intel Core 2 Duo processor running at 2.66 GHz) and the following well-known benchmark ontologies.

Lehigh University Benchmark (LUBM) [7] is a de facto standard for measuring performance of Semantic Web systems. The benchmark consists of a relatively simple ontology that describes the university domain, an ABox generator that is parameterised with the number of universities for which the data is to be generated, and 14 queries. In our experiments, we used just one university: this ABox contains about 100,000 triples, which is roughly of the size that our reasoner was designed to handle.

VICODI is an ontology about European history that was manually developed in the EU-funded VICODI project.[1] The ontology contains a relatively small and simple TBox; however, the ABox is relatively large, with many interconnected individuals.

[1]http://www.vicodi.org/

Table 1: Test Ontologies

	TBox axioms	Class assertions	Property assertions	DL expressivity
VICODI	223	33,238	82,943	$\mathcal{ALHI}(\mathbf{D})$
SEMINTEC	219	17,941	47,299	$\mathcal{ALHIF}$
LUBM	93	18,128	82,415	$\mathcal{ALEHI}^+(\mathbf{D})$

Table 2: Performance Test Results

	Loading	Initial reasoning	Incremental reasoning
VICODI	2127	2820	156
SEMINTEC	1048	1123	157
LUBM	2597	818	135

SEMINTEC is an ontology about financial services created in the SEMINTEC project at the University of Poznan.[2] Like VICODI, the TBox of the ontology is relatively simple, but the ABox is large. The main difference between VICODI and SEMINTEC is that the former does not use the equality predicate (*owl*:*sameAs*), whereas the latter does, and it is known that the presence of equality can significantly affect the performance of Semantic Web reasoners.

Table 1 presents some statistical information about the ontologies used in the evaluation.

For each ontology, we conducted the following tests. First, we loaded the ontology into the reasoner. Second, we instructed the reasoner to compute the materialisation. Third, we updated the ABox and instructed the reasoner to adjust the materialisation. The sets of added and removed assertions were selected as follows. We selected at random an individual in the data set. Starting from this individual, we traversed the relationships in the data set until we gathered a sample of 1,000 assertions. All of the selected assertions were scheduled for deletion. Furthermore, we renamed all the individuals in the selected sample (i.e., we replaced each individual with a fresh name), and we scheduled assertions for insertions. These are rather large updates relative to what we expect in the IMP setting, but we are of course using much more capable hardware than what is available in a typical mobile device.

The times (in milliseconds) necessary to accomplish these three tasks on our test ontologies are summarised in Table 2. As one can see, all actions can be completed in under a second, and incremental reasoning is particularly effective.

7. CONCLUSION

In this paper we have described Delta-Reasoner—a key component of the Intelligent Mobile Platform for supporting context-aware applications for mobile devices. Context-aware applications and the mobile platform impose unusual requirements on the reasoner, and these have been met by novel design features, including extensive support for built-ins, incremental reasoning, and continuous querying. Although we have so far been able to conduct only a very preliminary evaluation, the results are very encouraging, with sub-second query response times even after relatively large-scale changes to the data.

[2]http://www.cs.put.poznan.pl/alawrynowicz/semintec.htm

8. REFERENCES

[1] *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2nd edition, August 2007.

[2] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proc. of the 1st Int. Semantic Web Conf. (ISWC 2002)*, volume 2342 of *LNCS*, pages 54–68, Sardinia, Italy, June 9–12 2002. Springer.

[3] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the Semantic Web Recommendations. In *Proc. of the 13th Int. Conf. on World Wide Web (WWW 2004)—Alternate Track*, pages 74–83, New York, NY, USA, May 17–20 2004. ACM.

[4] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *Proc. of the 31st Int. Conf. on Very Large Data Bases (VLDB 2005)*, pages 1216–1227, Trondheim, Norway, August 30–September 2 2005.

[5] B. Glimm, I. Horrocks, C. Lutz, and U. Sattler. Conjunctive Query Answering for the Description Logic $\mathcal{SHIQ}$. *Journal of Artificial Intelligence Research*, 31:151–198, 2008.

[6] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: Combining logic programs with description logic. pages 48–57. ACM, 2003.

[7] Y. Guo, Z. Pan, and J. Heflin. An Evaluation of Knowledge Base Systems for Large OWL Datasets. In *Proc. of the 3rd Int. Semantic Web Conference (ISWC 2004)*, volume 3298 of *LNCS*, pages 274–288, Hiroshima, Japan, November 7–11 2004. Springer.

[8] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD 1993)*, pages 157–166, Washington, DC, USA, May 26–28 1993. ACM.

[9] V. Haarslev and R. Möller. RACER System Description. In *Proc. of the 1st Int. Joint Conf. on Automated Reasoning (IJCAR 2001)*, volume 2083 of *LNAI*, pages 701–706, Siena, Italy, June 18–23 2001. Springer.

[10] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML, W3C Member Submission, May 21 2004.

[11] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax, February 10 2004.

[12] Marko Luther, Yusuke Fukazawa, Matthias Wagner, and Shoji Kurakake. Situational reasoning for task-oriented mobile service recommendation. *Knowledge Eng. Review*, 23(1):7–19, 2008.

[13] B. Motik, B. Cuenca Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. OWL 2 Web Ontology Language: Profiles, W3C Recommendation, October 27 2009.

[14] B. Motik, P. F. Patel-Schneider, and B. Parsia. OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax, W3C Recommendation, October 27 2009.

[15] B. Motik, R. Shearer, and I. Horrocks. Hypertableau Reasoning for Description Logics. *Journal of Artificial Intelligence Research*, 36:165–228, 2009.

[16] B. Parsia and E. Sirin. Pellet: An OWL-DL Reasoner. Poster at the 3rd Int. Semantic Web Conference (ISWC 2004), November 7–11 2004.

[17] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF, W3C Recommendation, January 15 2008.

[18] M. Staudt and M. Jarke. Incremental Maintenance of Externally Materialized Views. In *Proc. of the 22th Int. Conf. on Very Large Data Bases (VLDB '96)*, pages 75–86, Mumbai, India, September 3–6 1996. Morgan Kaufmann.

[19] D. Tsarkov and I. Horrocks. FaCT++ Description Logic Reasoner: System Description. In *Proc. of the 3rd Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *LNAI*, pages 292–297, Seattle, WA, USA, August 17–20 2006. Springer.

[20] R. Volz, S. Staab, and B. Motik. Incrementally Maintaining Materializations of Ontologies Stored in Logic Databases. *Journal of Data Semantics II*, 3360:1–34, 2005. LNCS, Springer.

[21] Xiaohang Wang, Daqing Zhang, Tao Gu, and Hung Keng Pung. Ontology based context modeling and reasoning using owl. In *PerCom Workshops*, pages 18–22. IEEE Computer Society, 2004.

[22] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[23] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proc. of the 1st Int. Workshop on Semantic Web and Databases (SBWCB 2003)*, pages 131–150, Berlin, Germany, September 7-8 2003.