A Cross-Platform Benchmark Framework for Mobile Semantic Web Reasoning Engines

William Van Woensel, Newres Al Haider, Ahmad Ahmad, and Syed S.R. Abidi

NICHE Research Group, Faculty of Computer Science,
Dalhousie University, Halifax, Canada
{william.van.woensel,newres.al.haider,ahmad.ahmad,raza.abidi}@dal.ca

Abstract. Semantic Web technologies are used in a variety of domains for their ability to facilitate data integration, as well as enabling expressive, standards-based reasoning. Deploying Semantic Web reasoning processes directly on mobile devices has a number of advantages, including robustness to connectivity loss, more timely results, and reduced infrastructure requirements. At the same time, a number of challenges arise as well, related to mobile platform heterogeneity and limited computing resources. To tackle these challenges, it should be possible to benchmark mobile reasoning performance across different mobile platforms, with rule- and datasets of varying scale and complexity and existing reasoning process flows. To deal with the current heterogeneity of rule formats, a uniform rule- and data-interface on top of mobile reasoning engines should be provided as well. In this paper, we present a cross-platform benchmark framework that supplies 1) a generic, standards-based Semantic Web layer on top of existing mobile reasoning engines; and 2) a benchmark engine to investigate and compare mobile reasoning performance.

Keywords: Semantic Web, benchmarks, software framework, rule-based reasoning, SPIN.

1 Introduction

By supplying a formal model to represent knowledge, Semantic Web technology facilitate data integration as well as expressive rule-based reasoning over Web data. For example, in the healthcare domain, the use of specialized, Semantic Web medical ontologies facilitate data integration between heterogeneous data sources [10], while Semantic Web reasoning processes are employed to realize Clinical Decision Support Systems (CDSS) [21,6].

Reflecting the importance of reasoning in the Semantic Web, a range of rule languages and reasoning engine implementations, using an assortment of reasoning techniques, are available. Such reasoners range from Description Logic (DL)-based reasoners relying on OWL ontology constraints [17] to general-purpose reasoners, supporting a variety of rule languages (e.g., RuleML [7],

P. Mika et al. (Eds.) ISWC 2014, Part I, LNCS 8796, pp. 389–408, 2014.
© Springer International Publishing Switzerland 2014

SWRL [20] and SPIN [24]) and relying on different technologies, including Prolog (e.g., XSB[1]), deductive databases (e.g., OntoBroker[2]) and triple stores (e.g., Jena[3]). In general, rule-based reasoning techniques, as for instance used in decision support systems, allow a clear separation between domain knowledge and application logic. Consequently, domain knowledge can be easily edited, updated and extended without the need to disrupt the underlying system.

Up until now, knowledge-centric reasoning systems are typically developed for deployment as desktop or server applications. With the emergence of mobile devices with increased memory and processing capabilities, a case can be made for mobile reasoning systems. In fact, mobile RDF stores and query engines are already available, including RDF On the Go [25], AndroJena[4], i-MoCo [32], and systems such as MobiSem [33]. As such, a logical next step is to deploy rule-based reasoning, an essential part of the Semantic Web, on mobile devices as well.

Deploying mobile reasoning processes, as opposed to relying on remote services, has a number of advantages. In particular, local reasoning support allows making timely inferences, even in cases where connectivity is lacking. This is especially important in domains such as healthcare, where non- (or too late) raised alerts can negatively impact the patient's health. Secondly, given the myriad of data that can be collected about mobile users, privacy issues can play a role. A mobile user could (rightly) be uncomfortable with sharing certain information outside of the mobile device, for instance in context-aware [29] and mobile health scenarios [2,19]. By deploying reasoning processes locally, no privacy-sensitive data needs to be wirelessly communicated, while the advantages of rule-based reasoning is still accessible to mobile apps.

Performing mobile reasoning gives rise to challenges as well, both related to mobile device and platform heterogeneity as well as limited device capabilities. Furthermore, it is clear that each system has its own particular requirements regarding reasoning [13], which determine the complexity and scale of the rule- and dataset, as well as the particular reasoning process flow. In light of mobile device limitations, this makes it paramount to supply developers with the tools to benchmark, under their particular reasoning setup, different mobile reasoning engines. This way, developers may accurately study the performance impact of mobile deployment, and identify the best reasoning engine for the job. For instance, this may inform architecture decisions where reasoning tasks are distributed across the server and mobile device based on their complexity [2]. In addition, considering the fragmented mobile platform market (with systems including Android, iOS, Windows Phone, BlackBerry OS, WebOS, Symbian, ..), it should be straightforward to execute the same benchmark setup across multiple mobile platforms.

Compounding the problem of mobile benchmarking, current freely and publicly available mobile reasoning solutions support a variety of different rule and

[1] http://xsb.sourceforge.net/
[2] http://www.semafora-systems.com/en/products/ontobroker/
[3] http://jena.sourceforge.net/
[4] http://code.google.com/p/androjena/

data formats. In fact, the heterogeneity of rule languages is a general problem among rule-based reasoners [26]. We also note that multiple Semantic Web rule standards are currently available as well (e.g., RuleML, SWRL, SPIN). To avoid developers having to re-write their rule- and dataset to suit each engine, a single rule and data interface should be available. For our purposes, the most interesting rule language is SPIN, a W3C Member Submission based on the SPARQL query language. SPARQL is well-known and understood by most Semantic Web developers, reducing the learning threshold compared to other alternatives.

In this paper, we present a cross-platform Benchmark Framework for mobile Semantic Web reasoning engines. As its main goal, this framework aims to empower developers to investigate and compare mobile reasoning performance in their particular reasoning setups, using their existing standards-based ruleset and dataset. This framework comprises two main components:

- A generic, standards-based **Semantic Web Layer** on top of mobile reasoning engines, supporting the SPIN rule language. Behind the scenes, the supplied ruleset (SPIN) and dataset (RDF) are converted to the custom rule and data formats of the various supported reasoning engines.
- A **Benchmark Engine** that allows the performance of the different reasoning engines to be studied and compared. In this comparison, any existing domain-specific rulesets and datasets of varying scale and complexity can be tested, as well as different reasoning process flows.

By realizing this framework as a *cross-platform* solution, the same benchmarks can be easily applied across different mobile platforms. The framework is implemented in JavaScript using the PhoneGap[5] development tool, which allows mobile web apps to be deployed as native apps on a multitude of platforms (e.g., Android, iOS) . As a result, our framework allows benchmarking both JavaScript and native systems. The framework further has an *extensible* architecture, allowing new rule/data converters, reasoning flows and engines to be easily plugged in. Finally, we present an *example benchmark* in an existing clinical decision support scenario, to serve as a proof-of-concept and to investigate mobile reasoning performance in a real-world scenario. Our **online documentation** [31], associated with the presented benchmark framework, links to the source code and contains detailed instructions on usage and extension as well (these docs are referenced throughout the paper).

This paper is structured as follows. In Section 2, we discuss relevant background. Section 3 elaborates on the Mobile Benchmark Framework architecture and its main components. We continue by summarizing the measurement criteria (Section 4) and how developers can use the framework (Section 5). Section 6 summarizes the results of the example benchmark. In Section 7, we present related work, and Section 8 presents conclusions and future work.

[5] `http://phonegap.com/`

2 Background

2.1 Semantic Web Reasoning

An important aspect of the Semantic Web is reasoning, whereby reasoners may exploit the assigned semantics of OWL data, as well as the added expressivity given by domain-specific rules and constraints. Current semantic rule standards include the Semantic Web Rule Language (SWRL) [20], Web Rule Language (WRL) [3], Rule Markup/Modeling Language (RuleML) [7] and SPARQL Inferencing Notation (SPIN) [24]. In addition, many reasoning engines also introduce custom rule languages (e.g., Apache Jena[6]). Clearly, this multitude of semantic rule languages prevent the direct re-use of a single ruleset when benchmarking. To tackle this problem, our benchmark framework supplies a generic Semantic Web layer across the supported rule engines, supporting SPIN as the input rule language.

SPIN (SPARQL Inferencing Notation) is a SPARQL-based rule- and constraint language. At its core, SPIN provides a natural, object-oriented way of dealing with constraints and rules associated with RDF(S)/OWL classes. In the object-oriented design paradigm, classes define the structure of objects (i.e., attributes) together with their behavior, including creating / changing objects and attributes (rules) as well as ensuring a consistent object state (constraints). Reflecting this paradigm, SPIN allows directly associating locally-scoped rules and constraints to their related RDF(S)/OWL classes.

To represent rules and constraints, SPIN relies on the SPARQL Protocol and RDF Query Language (SPARQL) [14]. SPARQL is a W3C standard with well-formed query semantics across RDF data, and has sufficient expressivity to represent both queries as well as general-purpose rules and constraints. Furthermore, SPARQL is supported by most RDF query engines and graph stores, and is well-known by Semantic Web developers. This results in a low learning curve for SPIN, and thus also facilitates the re-encoding of existing rulesets to serve as benchmark input. In order to associate SPARQL queries with class definitions, SPIN provides a vocabulary to encode queries as RDF triples, and supplies properties such as `spin:rule` and `spin:constraint` to link the RDF-encoded queries to concrete RDF(S)/OWL classes.

2.2 Reasoning Engines

Below, we elaborate on the reasoning engines currently plugged into the Mobile Benchmark Framework.

AndroJena[7] is an Android-ported version of the well-known Apache Jena[8] framework for working with Semantic Web data. In AndroJena, RDF data can be directly loaded from a local or remote source into an RDF store called a *Model*, supporting a range of RDF syntaxes.

[6] `https://jena.apache.org/`
[7] `http://code.google.com/p/androjena/`
[8] `https://jena.apache.org/`

Regarding reasoning, AndroJena supplies an RDFS, OWL and rule-based reasoner. The latter provides both forward and backward chaining, respectively based on the standard RETE algorithm [12] and Logic Programming (LP). In addition, the reasoning engine supports a hybrid execution model, where both mechanisms are employed in conjunction[9,10]. Rules are specified using a custom rule language (which resembles a SPARQL-like syntax), and are parsed and passed to a reasoner object that is applied on a populated Model, which creates an *InfModel* supplying query access to the inferred RDF statements. Afterwards, new facts can be added to this InfModel; after calling the `rebind` method, the reasoning step can be re-applied.

RDFQuery[11] is an RDF plugin for the well-known jQuery[12] JavaScript library. RDFQuery attempts to bridge the gap between the Semantic Web and the regular Web, by allowing developers to directly query RDF (e.g., injected via RDFa [18]) gleaned from the current HTML page. RDF datastores can also be populated directly with RDF triples.

In addition to querying, RDFQuery also supports rule-based reasoning. Conditions in these rules may contain triple patterns as well as general-purpose filters. These filters are represented as JavaScript functions, which are called for each currently matching data item; based on the function's return value, data items are kept or discarded. The reasoning algorithm is "naïve", meaning rule are executed in turn until no more new results occur[13].

RDFStore-JS[14] is a JavaScript RDF graph store supporting the SPARQL query language. This system can be either deployed in the browser or a Node.js[15] module, which is a server-side JavaScript environment.

Comparable to AndroJena (see Section 2.2), triples can be loaded into an RDF store from a local or remote data source, supporting multiple RDF syntaxes. Regarding querying, RDFStore-JS supports SPARQL 1.0 together with parts of the SPARQL 1.1 specification. However, RDFStore-JS does not natively support rule-based reasoning. To resolve this, we extended the system with a reasoning mechanism that accepts rules as SPARQL 1.1 INSERT queries, whereby the WHERE clause represents the rule condition and the INSERT clause the rule result. This mechanism is naïve, executing each rule in turn until no more new results are inferred (cfr. RDFQuery).

Nools[16] is a RETE-based rule engine, written in JavaScript. Like RDFStore-JS, this system can be deployed both on Node.js as well as in the browser.

[9] `http://jena.apache.org/documentation/inference/#rules`
[10] Currently, we rely on the default configuration settings, which uses the hybrid execution model.
[11] `https://code.google.com/p/rdfquery/wiki/RdfPlugin`
[12] `http://jquery.com`
[13] The engine had to be extended to automatically resolve variables in the rule result.
[14] `http://github.com/antoniogarrote/rdfstore-js`
[15] `http://nodejs.org/`
[16] `https://github.com/C2FO/nools`

In contrast to the two other evaluated JavaScript systems, Nools presents a fully-fledged reasoning engine, supporting a non-naïve reasoning algorithm (RETE). Also, as opposed to the other evaluated systems, Nools does not natively support RDF. The engine is also used differently when performing reasoning. In case of Nools, a developer first supplies the rules, formulated using their custom rule language, in the form of a *flow*. The supplied flow is afterwards compiled into an internal representation (whereby pre-compilation can be applied to avoid repeating the compilation step each time). A *session* is an instance of the flow, containing the RETE working memory in which new facts are asserted. After creating and compiling the rule flow, the dataset is asserted in the session, after which the asserted data is matched to the defined rules.

Summary. Despite the potential of mobile reasoning processes, we observe a current lack of freely and publicly available mobile solutions. The above mentioned JavaScript engines were developed for use on either the server-side (using an environment such as Node.js) or a desktop browser, which makes their performance on mobile platforms uncertain. And similarly, while AndroJena represents port to the mobile Android platform, it is unclear to what extent the reasoning engine was optimized for mobile devices.

At the same time, our example benchmark (see Section 6), conducted in a real-world clinical decision support scenario, shows that these reasoning engines already have acceptable performance for small rule- and datasets. Moreover, our Mobile Benchmark Framework empowers developers to cope with this uncertainty of mobile performance, by allowing them to investigate the feasibility of locally deploying particular reasoning tasks. We further note that, as mobile device capabilities improve and demand for mobile reasoning deployment increases, more mobile-optimized reasoning engines are likely to become available. Recent efforts from the literature to optimize mobile reasoning processes in certain domains (i.e., context-awareness) have been already observed [29].

3 Mobile Benchmark Framework

In this section, we give an architecture overview of the Mobile Benchmark Framework. The framework architecture comprises two main components: 1) a generic **Semantic Web layer**, supplying a uniform, standards-based rule- and dataset interface to mobile reasoning engines; and 2) a **Benchmark Engine**, to investigate and compare mobile reasoning performance. Figure 1 shows the framework architecture.

During benchmark execution, the particular benchmark rule- and dataset (encoded in SPIN and RDF, respectively) are first passed to the generic Semantic Web layer. In this layer, a local component (called Proxy) contacts an external Conversion Web service, to convert the given rules and data into the formats supported by the plugged-in reasoning engines. In this Web service, conversion is performed by custom converters, each of which supports a particular rule or data format. Afterwards, the conversion results are returned to the Proxy and passed on to the Benchmark Engine.

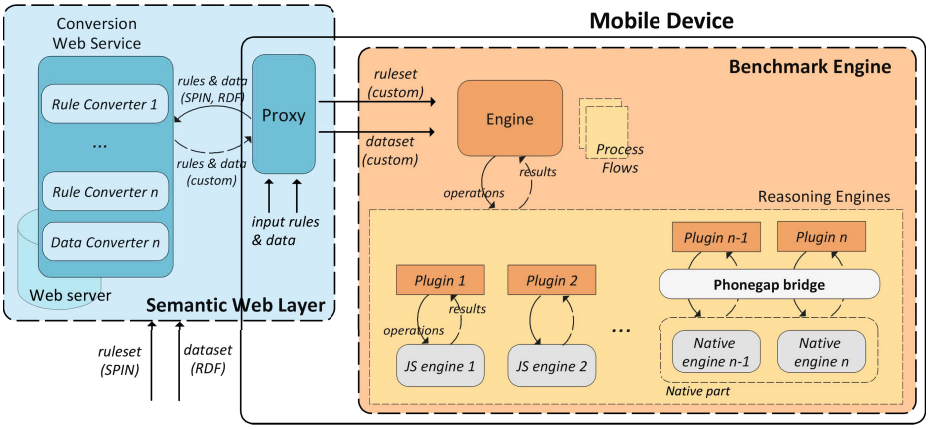

Fig. 1. Framework Architecture

In the Benchmark Engine, reasoning can be conducted using different process flows, to better align the benchmarks with actual, real-world reasoning systems (e.g., decision support). A particular reasoning flow is realized by invoking the uniform interface methods (e.g., load data, execute rules) of the benchmarked reasoning engine. Each mobile reasoning engine requires a plugin implementing this interface, which translates method invocations to the underlying reasoning engine. In case of native mobile reasoning engines, these plugins communicate with the native engine code over the PhoneGap communication bridge.

In the sections below, we elaborate on the two main architecture components.

3.1 Semantic Web Layer

This layer supplies a single, standards-based rule- and dataset interface for framework, allowing developers to cope with the heterogeneous rule- and dataset formats. Currently, the layer respectively supports SPIN[17] and RDF as input rule and data formats.

The conversion functionality is accessed via an intermediate JavaScript component called the Proxy, which comprises methods for rule and data conversion. Behind the scenes, the Proxy contacts a RESTful Conversion Web service (deployed on an external Web server) to perform the conversion tasks, thus introducing a layer of abstraction. We opted for a web service approach, since the only currently available SPIN API is developed for Java (by TopBraid [23]). The Conversion Web service utilizes the API to convert incoming SPIN rules into an Abstract Syntax Tree (AST). This AST is then analyzed by plugged-in converters, using the provided Visitor classes (Visitor design pattern), to convert the SPIN rules into equivalent rules[18] in other formats. In case data conversion

[17] The input SPIN rules do not need to be RDF-encoded.

[18] While only SPIN language features can be referenced in input rules, any (SPIN-encoded) core inferences should be mappable to a target IF-THEN rule format.

is required as well, a data converter can utilize the Apache Jena library to deal with incoming RDF data.

Below, we shortly elaborate on the currently developed converters. Afterwards, we discuss how the layer can be extended with new converters.

AndroJena (see Section 2.2) defines its own custom rule language, which resembles a triple pattern-like syntax. As such, rule conversion to SPIN (which relies on the likewise triple pattern-based SPARQL) is relatively straightforward. Comparably, **RDFQuery** (see Section 2.2) utilizes triple patterns in rule definitions, facilitating rule conversion. To create the JavaScript filter functions, function strings are generated and returned, which are evaluated (using the JS `eval` command) by the JavaScript Proxy to obtain the actual function constructs. **RDFStore-JS** requires converting SPIN rules, which are represented as CONSTRUCT queries[19], to equivalent queries using the INSERT keyword from SPARQL 1.1/Update [14].

As mentioned, **Nools** (see Section 2.2) is the only reasoning engine under consideration without built-in Semantic Web support. At the same time however, their generic rule language supports domain-specific extensions, by allowing rule definitions to include custom data types (e.g., data type `Message`). These data types can then be instantiated in the incoming dataset, and referenced in the defined rules. To add Semantic Web support, we include custom RDFStatement, RDFResource, RDFProperty and RDFLiteral data types into rule definitions, and convert incoming SPIN rules to Nools rules referencing these data types. The incoming RDF dataset is converted to instances of these custom data types, and asserted as facts in the *session*.

Currently, the converters support SPIN functions representing primitive comparators (greater, equal, ..), as well as logical connectors in FILTER clauses. Support for additional functions needs to be added to the respective converter classes. More advanced SPARQL query constructs, such as (not-)exists, optional, minus and union, are currently not supported, since it is difficult to convert them to all rule engine formats, and they have not been required up until now by our real-world test rule- and datasets (e.g., see example benchmark in Section 6).

Extensibility. To plugin a new data- or rule-format, developers create a new converter class. Each converter class implements a uniform rule- (or data-) conversion interface, which accepts the incoming SPIN rules / RDF data and returns Strings in the correct rule / data format. Each converter class also defines a unique identifier for the custom format, since conversion requests to the Web service specify the target format via its unique identifier.

New converter classes need to be listed (i.e., package and class name) in a configuration file, which is read by the Web service to dynamically load converter class definitions. As such, converters can be easily plugged in without requiring alterations to the web service code. Our online documentation [31] contains more detailed instructions on developing new converters.

[19] `http://www.w3.org/Submission/2011/SUBM-spin-modeling-20110222/#spin-rules-construct`

3.2 Benchmark Engine

The Benchmark Engine performs benchmarks on mobile reasoning engines under particular reasoning setups, with the goal of investigating and comparing reasoning performances. Below, we first discuss currently supported reasoning setups; afterwards, we elaborate on the extensibility of this component.

Reasoning setups comprise the particular process flows via which reasoning may be executed. By supporting different setups (and allowing new ones to be plugged in), benchmarks can be better aligned to real-world reasoning systems. From our work in clinical decision support, we identified two general process flows:

Frequent Reasoning: In the first flow, the system stores all health measurements and observations (e.g., heart rate, symptoms), collectively called clinical facts, in a data store. To infer new clinical conclusions, frequent reasoning is applied to the entire datastore, comprising all collected clinical facts together with the patient's baseline clinical profile (e.g., including age, gender and ethnicity). Concretely, this entails loading a reasoning engine with the entire datastore each time a certain timespan has elapsed, and executing the relevant ruleset.

Incremental Reasoning: In the second flow, the system implements clinical decision support by applying reasoning each time a new clinical fact is entered. In this case, the reasoning engine is loaded with an initial baseline dataset, containing the patient's clinical profile and historical (e.g., previously entered) clinical facts. Afterwards, the engine is kept in memory, whereby new facts are dynamically added to the reasoning engine. After each add operation, reasoning is re-applied to infer new clinical conclusions[20].

It can be observed that the Frequent Reasoning process flow reduces responsiveness to new clinical facts, while also incurring a larger performance overhead since the dataset needs to be continuously re-loaded. Although the Incremental Reasoning flow improves upon responsiveness, it also incurs a larger consistent memory overhead, since the reasoning engine is continuously kept in memory. The most suitable flow depends on the requirements of the domain; for instance, Incremental Reasoning is a better choice for scenarios where timely (clinical) findings are essential. The Benchmark Engine enables developers to perform mobile reasoning benchmarking using process flows that are most suitable for their setting. We note that additional flows can be plugged in as well, as mentioned at the end of this Section.

In addition, the particular reasoning engine may dictate a particular process flow as well (see Section 2.2). For instance, in case of RDFQuery, RDFStore-JS and AndroJena, the data is first loaded into the engine and rules are subsequently executed (*LoadDataExecuteRules*). For Nools, rules are first loaded into the engine to compile the RETE network, after which the dataset is fed into the network and reasoning is performed (*LoadRulesDataExecute*).

[20] An algorithm is proposed in [16] to optimize this kind of reasoning, which is implemented by the reasoning engine presented in [29].

We note that the former type of process flow (i.e., Frequent and Incremental Reasoning) indicates the reasoning timing, and is chosen based on domain requirements; while the latter flow type defines the operation ordering, and is determined by the employed reasoning engine[21]. For a single benchmark, the *reasoning setup* thus comprises a combination of two flows of each type. Figures 2/A and B illustrate the possible setups for our current reasoning engines.

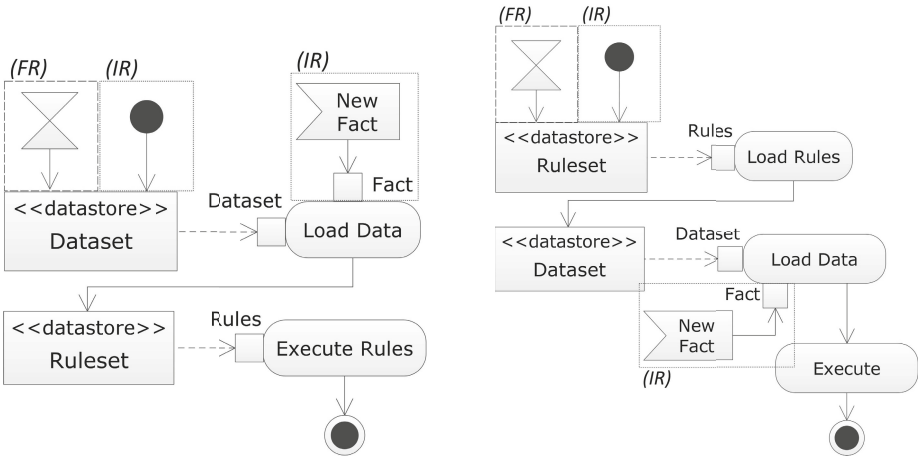

Fig. 2. (A) Frequent Reasoning and (B) Incremental Reasoning process flow (RDFQuery, RDFStore-JS, AndroJena)

Figure 2/A shows Frequent Reasoning (FR) and Incremental Reasoning (IR) for *LoadDataExecuteRules* (RDFQuery, RDFStore-JS and AndroJena), and Figure 2/B shows the same for *LoadDataRulesExecute* (Nools). For both diagrams, Frequent Reasoning entails going through the entire diagram each time a particular timespan has elapsed (time event). For Incremental reasoning, the system traverses the diagram from start to finish at startup time, and proceeds (from the indicated place) each time a new fact is received (receive signal event).

As mentioned, the Benchmark Engine is implemented in JavaScript and deployed as a native mobile app using the PhoneGap cross-platform development tool. We chose Android as the deployment platform since, to our knowledge, the only (publicly and freely available) native mobile reasoning engine (AndroJena, see Section 2.2) runs on that platform.

Extensibility. In the Benchmark Engine, each reasoning setup is represented by a JavaScript object. Its `runBenchmark` method invokes operations from the uniform reasoning engine interface (e.g., load data, execute rules) to realize its

[21] We also note that, for *LoadRulesDataExecute*, the Execute Rules step is separated into two steps.

particular process flows. The object is added to a folder called `setups` and listed in a `mapping.json` file, which maps combinations of process flows (e.g., *FrequentReasoning*, *LoadDataExecuteRules*) to their corresponding setup object.

A new mobile reasoning engine is plugged into the Benchmark Engine by writing a JavaScript "plugin" object. This object implements the uniform interface invoked by reasoning setups (see above) and translates method invocations to the underlying engine. In addition, each object specifies a unique engine ID, the rule- and dataformat accepted by the engine, as well as the process flow dictated by the engine (see Section 3.2). Each plugin object is put in a separate file and folder, both named after the reasoning engine id.

To insert *native* reasoning engines, developers implement the plugin on the native platform (e.g., Android), likewise implementing the uniform engine interface and specifying the aforementioned information. The Benchmark Engine comprises a native part (see Figure 1) to manage these plugins. In addition, developers add a dummy JavaScript plugin object for the engine, indicating the unique engine ID. Behind the scenes, the Benchmark Engine replaces this dummy object by a proxy component that implements communication with the native plugin. This setup is illustrated in the Benchmark Engine part of Figure 1.

More detailed instructions on developing Benchmark Engine extensions can be found in our online documentation [31].

4 Measurement Criteria

The Mobile Benchmark Framework allows studying and comparing the following metrics:

- *Data and rule loading times*: Time needed to load data and rules (if necessary) into the reasoning engine. Data loading time is commonly used in database benchmarks [9] as well as Semantic Web benchmarks [15,5]. Note that this time does not include converting the initial standards-based rule- and dataset to native formats.
- *Reasoning times*: Time needed to execute the rules on the dataset and infer new facts. Typically, database benchmarks capture the query response time as well, including Semantic Web benchmarks [15,4].

Ideally, and especially on mobile devices, these performance criteria would include memory consumption as well. However, it is currently not technically possible to automatically measure this criterium for all reasoning engines. Android Java heap dumps accurately measure the memory consumption of Android engines, but can only measure the entire memory size of the natively deployed web app (comprising the JavaScript reasoning engines). The Chrome DevTools remote debugging support[22] can only be employed to record heap allocations inside the mobile Chrome browser, and furthermore needs to be invoked manually. Other works also cite the accurate measuring of in-memory repository sizes as a difficult problem [15].

[22] `https://developers.google.com/chrome-developer-tools/docs/remote-debugging`

Some related works also investigate completeness and soundness of inferencing [15]. This criterium was presented in the context of OWL constraint-based reasoning, which typically serves to enrich data access where incomplete inferences may already be acceptable. In rule-based systems (e.g., (clinical) decision support systems), inferencing completeness is often a hard requirement. That said, each reasoning engine plugin in our framework outputs any inferred facts, allowing developers to check inferencing completeness.

Other works focus on benchmarking performance for reasoning types such as large joins, Datalog recursion and default negation [26]. Although these benchmarks are certainly useful, the goal of the Mobile Benchmark Framework is not to measure performance for such specific reasoning types, but instead to facilitate mobile reasoning benchmarking given a particular existing reasoning setup; including rule- and datasets and reasoning process flows.

Finally, we do not measure rule- and dataset conversion performance. The goal of the Semantic Web layer is to provide a uniform rule- and data-interface to facilitate benchmarking; the layer will not be included in actual mobile reasoning deployments.

5 Usage

This section gives a birds-eye view of how developers can utilize the framework. More detailed deployment and usage instructions for the framework, including the Conversion Web service, are available in our online documentation [31].

To perform benchmarks, developers provide a configuration file that specifies the reasoning setup and engine to be used, the number of runs, as well as the benchmark dataset and ruleset. By performing multiple runs of the same benchmark and calculating average execution times, the impact of background OS processes is minimized. Below, we show the configuration for our example benchmark (see Section 6):

```
{
    processFlow : 'frequent_reasoning',
        // options: frequent_reasoning, incremental_reasoning
    engine : 'AndroJena',

    nrRuns : 20,

    ruleSet : {
        path : "res/rules/af/benchmark.spin-rules",
        format : 'SPIN' // options: SPIN, native
    },

    // in case of 'incremental reasoning': include 'baseline'
    // & 'single-item' config under dataSet
    dataSet : {
        path : "res/data/af/25/benchmark.nt",
```

```
        format : 'RDF', // options: RDF, native
        syntax : 'N-TRIPLE'
            // options: RDF/XML, N-TRIPLE, TURTLE, TTL, N3, RDF/XML-ABBREV
    }
}
```

This configuration indicates the process flow (`processFlow`) and reasoning engine (`engine`) to be used in the benchmark, as well as the number of benchmark runs (`nrRuns`). The ruleset and dataset can either be provided respectively in SPIN / RDF or native format (i.e., the engine's natively supported format). In the non-native case, the framework automatically contacts the Semantic Web layer on-the-fly for conversion to the engine's native format. Alternatively, a script is available to convert rules and data beforehand, ruling out the need for connecting to the Web service during benchmarking.

6 Example Benchmark

In this section, we present an example benchmark that serves as a proof-of-concept of our Mobile Benchmark Framework. As an added goal, this benchmark aims to indicate the performance of the presented mobile reasoning engines for a real-world reasoning task, namely an existing clinical decision support scenario. Importantly, we note that the goal of this section is not to exhaustively compare the performance of the plugged-in mobile reasoning engines[23].

Below, we shortly elaborate on the benchmark domain (including the data- and ruleset), and indicate the utilized hardware. Afterwards, we summarize the benchmarking results.

6.1 Benchmark Domain

The benchmark data and ruleset are taken from ongoing work on the Integrated Management Program Advancing Community Treatment of Atrial Fibrillation (IMPACT-AF) project [22]. IMPACT-AF aims to provide web- and mobile-based clinical decision support tools for primary care providers and patients, with the goal of better managing Atrial Fibrillation (AF). To improve the timeliness of clinical alerts and increase robustness to connectivity loss, this project includes outfitting a mobile app, used by patients to enter health measurements and observed symptoms, with local reasoning support.

The mobile **ruleset**, employed in this benchmark, represents part of the computerized guidelines for the treatment of Atrial Fibrillation, given by the Canadian Cardiovascular Society [8] and European Society of Cardiology [11]. The ruleset encompasses a total of 10 rules. An AF patient's **dataset** comprises health factors related to AF, including clinically relevant personal info (e.g., age, gender) and health measurements (e.g., blood pressure), as well as AF-specific symptoms and the International Normalized Ratio (INR). Collectively,

[23] A second paper, performing such a comparison for the same domain, is currently under review.

we refer to these data items as clinical facts. We generated benchmark datasets containing the above described clinical data, whereby clinical facts were created based on ranges encompassing both clinically normal values as well as abnormal ones. With the goal of investigating mobile reasoning scalability, our benchmarks consider a sequence of datasets, each containing an increasing amount of data. Each dataset triggers 40-50% of the rules.

The rule- and dataset of this benchmark, as well as instructions on how to run it, can be found in the online documentation [31] (for the latter, see the Usage part).

6.2 Hardware

The benchmarks were performed on a Samsung Galaxy SIII (model number GT-I9300), with a 1.4GHz quad-core processor, 1GB RAM and 16GB storage. The installed Android OS was version 4.3 (Jelly Bean) with API level 18.

6.3 Results

In Section 3.2, we described two main process flows to realize mobile reasoning, including Incremental Reasoning and Frequent Reasoning. Below, we separately summarize and discuss the results for each process flow.

Frequent Reasoning. Table 1 shows the average loading and reasoning times for each engine and for increasing dataset sizes. Each run of this flow involves loading the reasoning engine with the entire dataset (*load* column) and then executing the rules (*execute* column); the *total* column shows the total time of each run.

We note that for Nools, loading times also include loading the rules into the engine[24], in order to build the internal RETE network (data loading time is shown separately between parenthesis). For some engines, the reasoning step includes creating rule objects as well; since this rule creation step turned out to be trivial (never exceeding 50 ms), these times were added to the overall reasoning times.

Incremental Reasoning. In Table 2, we again summarize average loading and reasoning times for increasing sizes of the dataset. In this process flow, the reasoning engine is initially loaded with a baseline dataset (typically at startup time). As baseline dataset, we employed the dataset containing 25 clinical facts (1673 triples). A single run of this flow involves loading an additional fact into the engine (*load* column) and performing the execution step (*execute* column). The *total* column shows the total time of a single run. We refer to Table 1 for times on the initial loading of the baseline dataset.

[24] This time remains constant for increasing dataset sizes.

Table 1. Frequent Reasoning: Loading & Reasoning times for increasing dataset sizes (ms)

#triples	RDFQuery			RDFStore-JS			Nools			AndroJena		
	load	exec	**total**	load	exec	**total**	load	exec	**total**	load	exec	**total**
137	95	154	**249**	196	985	**1181**	8411 (560)	52	**8463**	94	104	**198**
393	230	506	**736**	750	1523	**2273**	9256 (1245)	88	**9344**	160	138	**298**
713	362	1165	**1527**	1269	1479	**2748**	10061 (2521)	78	**10139**	439	466	**905**
1673	673	6294	**6967**	2468	1606	**4074**	14707 (7399)	58	**14765**	560	3205	**3765**
3273	1348	36603	**37951**	4269	2145	**6414**	25580 (18731)	64	**25644**	1036	24921	**25957**
4873	1680	106212	**107892**	5592	2496	**8088**	49465 (41845)	358	**49823**	1509	79699	**81208**

Table 2. Loading & Reasoning times for a single fact (ms)

	RDFQuery	RDFStore-JS	Nools	AndroJena
load	42	8	22	16
execute	5941	1677	19	3426
total	**5983**	**1685**	**41**	**3442**

6.4 Discussion

In this section, we shortly discuss the benchmark results summarized above for each reasoning process flow.

Frequent Reasoning. Table 1 shows the Nools data loading time is problematic for larger (> 713 triples) datasets (the rule loading time is constant and averages ca. 7-8s). Regarding loading times, especially RDFQuery and AndroJena perform well (< 1s) for medium datasets (< 3273 triples), whereby AndroJena has the best loading performance overall.

At the same time, we note that AndroJena and RDFQuery, while performing well for smaller datasets, have a very problematic reasoning performance for larger datasets (≥ 1673 triples). Nools has by far the best reasoning performance, only exceeding 100ms for the largest dataset. Reasoning performance for RDFStore-JS remains reasonable, rising steadily as the datasets increase in size.

From the total times, we observe that **RDFStore-JS** is the most scalable solution for this particular process flow, performing best for larger datasets ($>$ 1673 triples). **AndroJena** is the better solution for smaller datasets (≤ 1673 triples).

It can be observed that the domain datasets are relatively small scale. Inside this limited scale however, the benchmark already identified clear differences in engine performance for increasing dataset sizes. For larger datasets, problematic mobile performance may for instance point the developer towards a distributed solution, combining local and server-side reasoning.

Also, we note the ruleset was not optimized to suit the employed reasoning mechanisms (e.g., RETE, Logic Programming) or dataset composition.

Investigating the effects of the various potential optimizations is beyond the scope of this paper, and will be considered in future work.

Incremental Reasoning. Table 2 shows that, as expected from the discussion above, Nools has by far the best performance in this reasoning step, with almost negligible reasoning times compared to the other engines. In contrast, reasoning times for the other three engines is comparable to their reasoning performance for this dataset size in the first process flow.

Consequently, we observe that, once the initial data and rule loading is out of the way, Nools has by far the best reasoning performance when incrementally adding facts in this process flow. As noted in the previous section, Nools data loading times for small datasets ($\leq$ 713 triples) are still acceptable (while we note that rule loading time will also decrease with the ruleset size). Therefore, **Nools** is the best option for this flow in case of *small datasets and rulesets*, since the low reasoning time makes up for the increased initialization time. In case *scalability* is required, **RDFStore-JS** remains the best option.

Conclusion. The above results indicate that, as expected, the most suitable engine depends on the target reasoning process flow, as well as the dataset (and ruleset) scale. At the same time however, we observe that scalability represents a serious issue for most mobile engines. We also note that, although taken from an existing, real-world clinical decision support system, the utilized ruleset is relatively straightforward, with for instance no rule chaining. If that had been the case, naïve reasoning mechanisms (as employed by RDFStore-JS and RDFQuery) would likely have a larger disadvantage compared to the fully-fledged AndroJena and Nools engines. If anything, this again indicates the importance of a Mobile Benchmark Framework that allows easily performing benchmarks with the particular rule- and dataset from the target use case.

7 Related Work

The Lehigh University Benchmark (LUBM) [15] supplies a set of test queries and a data generator to generate datasets, both referencing a university ontology. In addition, a test module is provided for carrying out data loading and query testing. This work aims to benchmark data loading and querying over large knowledge base systems featuring OWL / RDF(S) reasoning. The University Ontology Benchmark (UOBM) [27] builds upon this work, and extends it to support complete OWL-DL inferencing and improve scalability testing. Similarly, the Berlin SPARQL benchmark (BSBM) [5] supplies test queries, a data generator and test module for an e-commerce scenario. In this case, the goal is to compare performance of native RDF stores with SPARQL-to-SQL rewriters, and to put the results in relation with RDBMS.

The focus of the works presented above differs from our work, which is on the cross-platform benchmarking of mobile, rule-based Semantic Web reasoners; and facilitating such benchmarks by providing a uniform interface across different, natively supported rule- and data formats.

OpenRuleBench [26] is a suite of benchmarks for comparing and analyzing the performance of a wide variety of rule engines, spanning 5 different technologies and 11 systems in total. These benchmarks measure performance for types of reasoning such as large joins and Datalog recursion, targeting engines deployed on the desktop- and server-side. Instead, we focus on benchmarking Semantic Web reasoners deployed on mobile platforms. Additionally, we supply the tools for developers to benchmark their existing reasoning setup, including their rule- and dataset and particular reasoning flow.

The Intelligent Mobile Platform (IMP) supplies context-aware services to third-party mobile apps, and relies on the Delta Reasoner [29] to determine current context and identify appropriate actions. To cope with the particular requirements of context-aware settings, including the dynamicity of sensor data and the necessity of push-based access to context data, the Delta Reasoner implements features such as incremental reasoning and continuous query evaluation. However, this reasoner is currently not publicly available; and the integration of this reasoner into the mobile IMP still seems a work in progress.

8 Conclusions and Future Work

In this paper, we introduced a Mobile Benchmark Framework for the investigation and comparison of mobile Semantic Web reasoning engine performances. This framework was realized as a *cross-platform* solution, meaning a particular benchmark setup can be easily applied across mobile platforms. Furthermore, there is a strong focus on *extensibility*, allowing new rule- and data converters, reasoning process flows and engines to be plugged in. Throughout the paper, we indicated where and how extensions can be made by third-party developers.

An important goal of the framework is to empower developers to benchmark different reasoning engines, using their own particular reasoning setups and standards-based rule-and datasets. To that end, the framework comprises two main components:

- A generic, standards-based **Semantic Web Layer** on top of mobile reasoning engines, supporting the SPIN rule language. Given a standards-based ruleset (SPIN) and dataset (RDF), a conversion component returns this rule- and dataset transformed into the custom formats supported by the mobile reasoning engines.
- A **Benchmark Engine** that allows the performance of the different reasoning engines to be studied and compared. In this comparison, any domain-specific rule- and dataset with varying scale and complexity can be tested, as well as multiple reasoning process flows.

As a proof-of-concept, an example benchmark was performed using the framework, based on an existing clinical decision support system. Additionally, this benchmark aimed to measure the performance of mobile reasoning engines for such a real-world reasoning setup; and thus study the feasibility of locally deploying reasoning processes at this point in time. Although most benchmarked reasoning engines were not optimized for mobile use, the benchmark showed these

engines already feature reasonable performance for limited rule- and datasets. At the same time, we note scalability is certainly an issue, with the most efficient overall execution times for Frequent Reasoning rising to ca. 8s for the largest dataset (comprising 4873 triples). To support larger-scale setups, it is clear that much more work is needed to optimize rule-based Semantic Web reasoners for mobile deployment. Interest in performing such optimization has been observed recently in the literature [29], and is likely to increase as demand for mobile reasoning processes increases (e.g., from domains such as health care [1,2,19]).

Future work includes benchmarking mobile reasoning engines with rulesets of increased complexity. Support for additional SPIN functions and more advanced SPARQL constructs should be added to the Semantic Web layer, as these will likely be needed by such complex rulesets. Moreover, different optimization techniques will be applied as well to systematically evaluate their impact on performance. A number of relevant techniques can be utilized for this purpose, for instance based on RETE [28] or borrowed from SPARQL query optimization [30].

Currently, we are employing one of the mobile reasoning engines in the IMPACT-AF mobile app (see Section 6.1), where it features sufficient performance for the current rule- and dataset. As requirements for local reasoning increase, it is possible we will investigate custom optimizations to these engines for mobile deployment.

Acknowledgments. This research project is funded by a research grant from Bayer Healthcare.

References

1. Abidi, S.R., Abidi, S.S.R., Abusharek, A.: A Semantic Web Based Mobile Framework for Designing Personalized Patient Self-Management Interventions. In: Proceedings of the 1st Conference on Mobile and Information Technologies in Medicine, Prague, Czech Republic (2013)
2. Ambroise, N., Boussonnie, S., Eckmann, A.: A Smartphone Application for Chronic Disease Self-Management. In: Proceedings of the 1st Conference on Mobile and Information Technologies in Medicine, Prague, Czech Republic (2013)
3. Angele, J., Boley, H., De Bruijn, J., Fensel, D., Hitzler, P., Kifer, M., Krummenacher, R., Lausen, H., Polleres, A., Studer, R.: Web Rule Language (2005), `http://www.w3.org/Submission/WRL/`
4. Becker, C.: RDF Store Benchmarks with DBpedia comparing Virtuoso, SDB and Sesame (2008), `http://wifo5-03.informatik.uni-mannheim.de/benchmarks-200801/`
5. Bizer, C., Schultz, A.: The berlin sparql benchmark. International Journal on Semantic Web and Information Systems-Special Issue on Scalability and Performance of Semantic Web Systems (2009)
6. Blomqvist, E.: The use of semantic web technologies for decision support - a survey. Semantic Web 5(3), 177–201 (2014)

7. Boley, H., Tabet, S., Wagner, G.: Design rationale of RuleML: A markup language for semantic web rules. In: Cruz, I.F., Decker, S., Euzenat, J., McGuinness, D.L. (eds.) Proc. Semantic Web Working Symposium, pp. 381–402. Stanford University, California (2001)
8. Canadian Cardiovascular Society: Atrial Fibrillation Guidelines, http://www.ccsguidelineprograms.ca
9. Cattell, R.G.G.: An engineering database benchmark. In: The Benchmark Handbook, pp. 247–281 (1991), http://dblp.uni-trier.de/db/books/collections/gray91.html#Cattell91
10. Cheung, K.H., Prud'hommeaux, E., Wang, Y., Stephens, S.: Semantic web for health care and life sciences: a review of the state of the art. Briefings in Bioinformatics 10(2), 111–113 (2009)
11. European Society of Cardiology: Atrial Fibrillation Guidelines, http://www.escardio.org/guidelines-surveys/esc-guidelines/guidelinesdocuments/guidelines-afib-ft.pdf
12. Forgy, C.L.: Rete: A fast algorithm for the many patterns/many objects match problem. Artif. Intell. 19(1), 17–37 (1982)
13. Gray, J.: The Benchmark Handbook for Database and Transaction Systems. Morgan Kaufmann (1993)
14. Group, W.S.W.: SPARQL 1.1 Overview, W3C Recommendation (March 21, 2013), http://www.w3.org/TR/sparql11-overview/
15. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web 3(2), 158–182 (2005)
16. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD 1993, pp. 157–166. ACM, New York (1993), http://doi.acm.org/10.1145/170035.170066
17. Haarslev, V., Möller, R.: Description of the racer system and its applications. In: Goble, C.A., McGuinness, D.L., Möller, R., Patel-Schneider, P.F. (eds.) Description Logics. CEUR Workshop Proceedings, vol. 49 (2001)
18. Herman, I., Adida, B., Sporny, M., Birbeck, M.: RDFa 1.1 Primer, 2nd edn (2013), http://www.w3.org/TR/xhtml-rdfa-primer/
19. Hommersom, A., Lucas, P.J.F., Velikova, M., Dal, G., Bastos, J., Rodriguez, J., Germs, M., Schwietert, H.: Moshca - my mobile and smart health care assistant. In: 2013 IEEE 15th International Conference on e-Health Networking, Applications Services (Healthcom), pp. 188–192 (October 2013)
20. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission (May 21, 2004), http://www.w3.org/Submission/SWRL/
21. Hussain, S., Raza Abidi, S., Raza Abidi, S.: Semantic web framework for knowledge-centric clinical decision support systems. In: Bellazzi, R., Abu-Hanna, A., Hunter, J. (eds.) AIME 2007. LNCS (LNAI), vol. 4594, pp. 451–455. Springer, Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-73599-1_60
22. Integrated Management Program Advancing Community Treatment of Atrial Fibrillation: Impact AF, http://impact-af.ca/
23. Knublauch, H.: The TopBraid SPIN API (2014), http://topbraid.org/spin/api/
24. Knublauch, H., Hendler, J.A., Idehen, K.: SPIN - Overview and Motivation (2011), http://www.w3.org/Submission/spin-overview/

25. Le-Phuoc, D., Parreira, J.X., Reynolds, V., Hauswirth, M.: RDF On the Go: An RDF Storage and Query Processor for Mobile Devices. In: 9th International Semantic Web Conference, ISWC 2010 (2010)
26. Liang, S., Fodor, P., Wan, H., Kifer, M.: Openrulebench: An analysis of the performance of rule engines. In: Proceedings of the 18th International Conference on World Wide Web, pp. 601–610. ACM, New York (2009), http://doi.acm.org/10.1145/1526709.1526790
27. Ma, L., Yang, Y., Qiu, Z., Xie, G., Pan, Y., Liu, S.: Towards a complete owl ontology benchmark. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 125–139. Springer, Heidelberg (2006), http://dx.doi.org/10.1007/11762256_12
28. Matheus, C.J., Baclawski, K., Kokar, M.M.: Basevisor: A triples-based inference engine outfitted to process ruleml and r-entailment rules. In: Second International Conference on Rules and Rule Markup Languages for the Semantic Web, pp. 67–74 (November 2006)
29. Motik, B., Horrocks, I., Kim, S.M.: Delta-reasoner: A semantic web reasoner for an intelligent mobile platform. In: Proceedings of the 21st International Conference Companion on World Wide Web, WWW 2012 Companion, pp. 63–72. ACM, New York (2012), http://doi.acm.org/10.1145/2187980.2187988
30. Schmidt, M., Meier, M., Lausen, G.: Foundations of sparql query optimization. In: Proceedings of the 13th International Conference on Database Theory, ICDT 2010, pp. 4–33. ACM, New York (2010), http://doi.acm.org/10.1145/1804669.1804675
31. Van Woensel, W.: Benchmark Framework Online Documentation (2014), https://niche.cs.dal.ca/benchmark_framework/
32. Weiss, C., Bernstein, A., Boccuzzo, S.: I-MoCo: Mobile Conference Guide Storing and querying huge amounts of Semantic Web data on the iPhone-iPod Touch. In: Semantic Web Challenge 2008 (2008)
33. Zander, S., Schandl, B.: A framework for context-driven RDF data replication on mobile devices. In: Proceedings of the 6th International Conference on Semantic Systems, I-SEMANTICS 2010, pp. 22:1—-22:5. ACM, New York (2010), http://doi.acm.org/10.1145/1839707.1839735