Collaborative Benchmarking Rule-Reasoners with B-Runner

Federico Ulliana, Pierre Bisquert, Akira Charoensit, Renaud Colin, Florent Tornil, Quentin Yeche

Collaborative Benchmarking Rule-Reasoners with B-Runner

Federico Ulliana[1][0000−0002−9192−9573], Pierre Bisquert[2][0000−0001−9418−5330], Akira Charoensit[1][0000−0002−2471−2040], Renaud Colin[2][0009−0003−8910−4083], Florent Tornil[1][0009−0005−8865−3747], and Quentin Yeche[2][0009−0008−3562−256X]

[1] Inria, LIRMM, Univ Montpellier, CNRS, Montpellier, France
{federico.ulliana,akira.charoensit,florent.tornil}@inria.fr
[2] INRAE, Montpellier, France
{pierre.bisquert,renaud.colin,quentin.yeche}@inrae.fr

Abstract. Conducting experimental analysis on rule reasoners is a mainstream task for validating novel algorithms and systems. Nevertheless, providing robust, verifiable, and reproducible experiments can still raise a sensible challenge. This paper introduces B-Runner, an open library for *collaborative benchmarking* focusing on the deployment of extended tests for *knowledge and rule-based systems* with low cost and high robustness. B-Runner reduces the benchmarking setup time while guaranteeing experiment repeatability. Also, it improves the scrutability of experimental protocols thereby enhancing the fairness of system comparisons.

1 The Benchmarking Issue

Scientific experiments are essential for validating and improving systems, but they come with numerous challenges. These can be classified depending on whether they are related to planning, reporting, or conducting experiments.

Planning consists in writing down all design choices for the experiments. Notably, the benchmarks to use, the competitors to consider, the testing environment (hardware/software) and the target measures.

Reporting includes retrieving and analyzing the results, then choosing the most meaningful data as well as the most informative and succinct graphical representations.

Conducting consists in faithfully implementing what has been planned. This includes setting up the test environment, coding (and verifying) the experimental protocols, and then running the tests while handling potential failures.

While all phases of experimental analysis can cause issues, the experiment conduction is perhaps the least accessible part. Accessibility here is intended as the *time and effort it takes a user to conduct a robust experimental analysis*. Of course, with high accessibility experimental analysis can be deeper, which directly translates to more thoughtful validation of novel approaches.

Conducting is also intimately related to *repeatability*. Repeatability means that another user equipped with the appropriate software and hardware can

repeat the same experiments [14]. Repeatability is easily attained when experiment conduction is simple, reliable, and documented. Conversely, it may be a burden to prepare a test suite and write sophisticated instructions on how to configure and run a benchmark starting from ad-hoc scripts, not to mention debugging them to make them portable on different systems. Experiment repeatability has become increasingly relevant during the last decades. Nowadays many conferences have established reproducibility[3] tracks.

Independent vs Collaborative Benchmarking

In this work, we argue that the underlying issue with benchmarking conduction and repeatability lies in the lack of a platform for *collaborative benchmarking*. Indeed, the most common benchmarking scenario is one where activities are conducted *independently*, which in practice results in a plethora of hard-to-maintain scripts and ad hoc methods for running tests. While this consideration may apply to many fields, we emphasize that our focus is on the case of benchmarking *knowledge and rule-based reasoners* for which we introduce a dedicated library.

Independent Benchmarking. Figure 1.a) illustrates a set of users (here A, B, C) conducting an experimental analysis to understand the performances of a given reasoning system (which may or may not have been coded by one of the users). The system supports multiple query answering (QA) tasks on a knowledge base. In particular, users B and C want to test the performances of QA via query rewriting while user A is interested in the performances of QA via the chase.

Let us explain this scenario in detail. A knowledge base (KB) is composed of a *factbase* and a set of *rules* modelling semantic constraints on the data (ontologies, data-dependencies, etc) used to both enrich it and ensure its consistency [15]. The reasoner can be commanded through an API. The API also offers a number of basic features. For instance, *loading* and *building* the knowledge base starting from plain data and/or data-integration mappings [10]. But also, using a certain type of internal *storage* for the data (graph, relational, triplestore, in-memory/disk, etc.). And, finally, being able to *evaluate* queries on a factbase (without considering the rules). The API also includes a number of advanced features. These includes techniques which accounts for rules using *saturation* (also called *chase*) and *query rewriting*. The chase approach essentially consists at extending the factbase with the result of the application of rules [6]. Query rewriting in contrast compiles the rules into the input query thereby yielding a *reformulated* query which provides the same answers (as the input rules and query) on any database [9]. Query answering via the chase consists in evaluating the input query on the saturated knowledge base. Query answering via rewriting consists in evaluating the reformulated query on the input data.[4]

[3] reproducibility is stricter than repeatability: not only it ensures that the experiment can be re-run, but that it also yields the same result.

[4] To illustrate, consider the factbase $F = \{P(a)\}$, the rule $\forall x.P(x) \rightarrow R(x)$, and the boolean query $Q = \exists y.R(y)$. The chase yields $\{P(a), R(a)\}$ where Q answers true. Query rewriting yields reformulation $\exists y.R(y) \vee P(y)$ answering true on F.

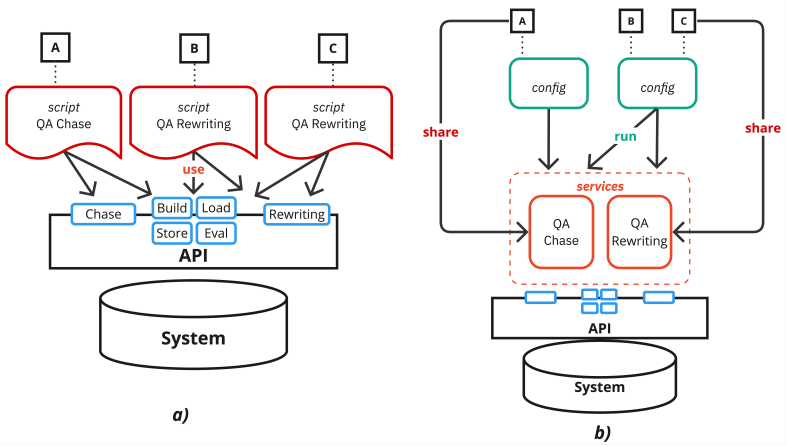

Fig. 1. Benchmarking: *a*) Independent vs *b*) Collaborative.

As Figure 1.a) shows, to conduct the test, every user writes a script coding an experimental protocol by leveraging on the API. Typically, this is done *from scratch*. Firstly, this is *time consuming*, as it requires learning the API and internals of the system as well as coding the measurements for the API calls. Secondly, this is *error prone*. Indeed, since scripts are *independently* written, for more complex tests it is not uncommon for the test protocols to even diverge, at the point measuring different operations.[5] All of this makes that setting up an experiment can have a significant cost and compromised robustness, resulting in low accessibility.

Collaborative Benchmarking. A principled solution to this issue is to provide users with a platform for collaborative-benchmarking, which simplifies the conduction of extended test trials. Collaborative benchmarking captures the idea that robust and repeatable experimental analysis can be achieved if *a community of users work together* on *i*) consolidating a set of *test protocols* which *ii*) can run on a *reliable framework* and *iii*) without *unnecessary complexity* at the user level. Figure 1.b) illustrates the case for collaborative benchmarking. Again, we consider users B and C testing QA via rewriting and user A testing QA via the chase. The first characteristic of this approach is that it allows users to share *benchmarkable services* (or simply services) coded in a common programming language, which very much improves protocol readability. Every such service implements a self-contained *testing protocol* which allows one to measure the performances (time, throughput, etc.) of a certain task. This test protocol is meant to be deployed on a variety of input scenarios and algorithm configurations - and not on a single one. By definition, sharing test implementations can benefit from code reviews in a collaborative environment, thus increasing

[5] typical errors are including/omitting optimizations and/or parsing time, writing on disk or standard output (logging, result export), improper cold/warm measures [14]

the robustness of the experimental analysis and avoiding the divergence of testing protocols. Figure 1.b) illustrates user A sharing a service for running QA via the chase, while user C shares a service for QA via rewriting. By fostering test reuse, user B can perform an experimental analysis without coding by reusing the service shared by C. This results in significant time savings. Besides test sharing, another key characteristic of this approach is that test trials are specified by users via *configurations*. This makes the specification of test trials more *declarative* than programmatic, and hence independent from any programming languages. In this aspect, collaborative benchmarking is strongly opposed to independent benchmarking, the latter allowing each script to be potentially written in a different language. Crucially, configuration files can also be shared, as illustrated for users B and C. Not only does this save time, but it allows users to *communicate* and *document* the content and aim of their experiments in a more standard and intelligible way.

1.1 Novelty and Contributions

We introduce B-Runner: *a Java tool for collaborative benchmarking on knowledge and rule-based reasoners.* The distinctive elements of the tool are the following.

1) Collaborative B-Runner allows users to share and reuse experimental protocols and test configurations. The notion of collaborative benchmarking has been little regarded for rule-based reasoners. To the best of our knowledge, B-Runner is the first attempt to systematize this approach for rule-based reasoners.

2) Simple and Robust B-Runner allows to define trial specifications through declarative configurations which require minimal coding and learning of test systems. B-Runner proposes a design pattern for writing test protocols that favors their scrutability. The execution of tests is controlled via the Java Microbenchmarking Harness (JMH) library. This favors converging towards robust error-free testing protocols and measures.

3) Extensible and Portable. B-Runner's generic architecture allows to easily include novel systems and testing protocols. B-Runner is Java-based, which makes it portable, yet still able to support benchmarking of non-Java systems.

1.2 User Groups and Paper Organization

B-Runner can be used by two types of users: *testers* and *providers*. Testers define experiments from available services using configuration files (for instance, user B in Figure 1.b). Providers share benchmarking services (for instance, users A and C in Figure 1.b that share services for testing respectively QA chase and QA rewriting). In the remainder of this paper, Section 2 presents how a tester can declare a benchmarking activity through a configuration file. Section 3 shows the provider side of offering a benchmarking protocol. Section 4 delves into features and limits of B-Runner.

2 Benchmarking in a Hurry!

The first key feature of B-Runner is the possibility of doing benchmarking at small cost for *tester* users. Let us define the benchmarking activities we consider. A benchmarking activity B is a sequence of service executions $B = (e_1, \ldots, e_k)$. Every service execution is a triple $e = (s, c, r)$ where s is a service, c is a configuration for the service, and r the repetition parameters. A configuration $c = (n, a, d)$ is a combination of an execution environment n on top of which the service s is executed by taking as input an algorithm configuration a and an input scenario d. The repetition parameter is a pair $r = (f, i)$ where f is the number of forks and i the number of the iterations for the execution of s given c.

```
#    Reasoner
reasoner = integraal

#    Scenarios (fixed rules and queries)
### 10M data
scenario.s10M.data              =   data10.dlgp
scenario.s10M.rules             =   ontology.dlgp
scenario.s10M.workload          =   queries.dlgp

### 100M data
scenario.s100M.data             =   data100.dlgp
scenario.s100M.rules            =   ontology.dlgp
scenario.s100M.workload         =   queries.dlgp

#    Algorithms
### chase (in-memory)
tool.inmem_chase.service        =   QAChase
tool.inmem_chase.dbType         =   inMemoryGraphStore
tool.inmem_chase.checker        =   semiOblivious

### rewriting (postgres)
tool.postgres_rew.service       =   QARewriting
tool.postgres_rew.driver        =   postgreSQL
tool.postgres_rew.driverURL     =   jdbc:postgresql://...

#    Environment
execution.basic_env.maxMemory   =   16g
execution.basic_env.timeout     =   10m
execution.basic_env.fork        =   2
execution.basic_env.iterations  =   3

#    Export
export = json
```

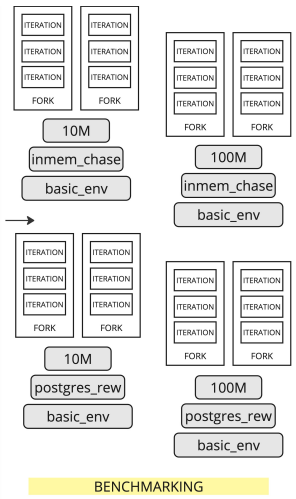

Fig. 2. Trial Configuration Example

Figure 2 illustrates a benchmarking activity made of four service executions (with their corresponding configurations and repetitions parameters). The goal of a *fork* is twofold. Firstly, it creates a "cold" execution environment. Secondly, it sets up the reasoning system for running a number of repetitions (of the same service). An *iteration* takes places within a fork, and consists in the actual execution of the service itself. It is worth pointing out that every fork in B-Runner triggers the creation of a new Java Virtual Machine (JVM) where the test runs. More specifically, this is accomplished by leveraging on Java Microbenchmark Harness (JMH), and happens to be useful to eliminate measure bias (due to cache, Just-In-Time compilation, etc.).

Figure 2 shows a configuration written by a tester and inspired by the example given in the introduction (Figure 1). The syntax represents a collection of properties which stands as a notation for *trees*.[6] In the example, reserved keywords for B-Runner are highlighted in blue. As indicated by line 2, the configuration applies to the InteGraal reasoner [3]. Keywords proper to the configuration of the reasoner are highlighted in red. The configuration then continues with the specification of two test scenarios "s10M" (lines 6-8) and "s100M" (lines 11-13). To declare different datasets, we use the reserved keyword `scenario`. Both scenarios consider the same ontology-rules and queries but differ in terms of data.[7] Then, two different algorithmic strategies for query answering are used. This is done with properties prefixed by the keyword `tool`. The first is via the chase procedure (line 17). Note also that the data is stored in a native InteGraal in-memory graph store (line 18) and that a specific variant of the chase, called *semi-oblivious* [7] is chosen (line 19). Similarly, a second strategy based on rewriting is defined (line 22). In this case, note that the data is stored on a (local or remote) PosgtreSQL server (line 23) which is accessible via the given connection URL (line 24). These options illustrate the richness and simplicity of the algorithm configuration that can be achieved. Finally, the execution environment for the tests is set. This is done to properties prefixed by the keyword `execution`. Options include setting the maximum JVM size to 16GB (line 27), setting the timeout for the execution of a *single* iteration to 10 minutes (line 28), and the number of forks and repetitions (lines 29-30). The last command (line 33) exports the results in JSON.

From this specification file, B-Runner automatically conducts a benchmarking activity on a sequence of configurations generated by combining *every* input scenario with *every* algorithm and *every* environment configuration. The resulting benchmarking activity is illustrated in Figure 2. Note that selective test execution (useful for example to restart failures) is also possible in B-Runner [1].

3 Sharing a New Benchmarkable Service

The second distinctive feature of B-Runner is the possibility of sharing new testing protocol for *provider* users. In contrast to configuration files, protocols must be *programmatically written*. B-Runner adopts Java, yet the methodology we present here is transposable to other programming languages.

In a nutshell, B-Runner proposes a design pattern for implementing protocols which aims at making a *service equivalent to an array of method references*. Below, we present a testing protocol for query answering via query rewriting, which measures time for each step of the task. Let us present the B-Runner API for specifying new testing protocols. (1) A new protocol can be specified by simply implementing the `serviceOperations` method. (2) In turn, this uses two methods for specifying the test steps: `setup` and `operation`. The method

[6] To illustrate, lines 6-8 of Figure 2 can be written in JSON as the record `{scenario: {s10M: {data:data10.dlgp, rule:ontology.dlgp, workload:queries.dlgp}}}`

[7] In the example, the .dlgp extension stands for the Datalog-Plus language [4].

`setup` corresponds to a step that has to be performed *for every fork*. The method `operation` corresponds to a step that has to be performed *for every iteration*.

```
1  public void serviceOperations() {
2
3    setup(DATA_LOADING,   this::setData);
4    setup(RULE_LOADING,   this::setRules);
5
6    operation(QUERY_LOADING,    this::setQuery);
7    operation(BUILD_REWRITER,   this::buildRewriter);
8    operation(QUERY_REWRITING,  this::rewrite);
9
10   operation(BUILD_EVALUATOR,  this::buildEvaluator);
11   operation(QUERY_EVALUATION, this::evalRewriting);
12 }
```

As the example illustrates, typical examples of setup steps include the loading of the scenario, notably data and rules which are considered fixed (lines 3-4). Then, operation steps include creating objects and executing query rewriting (lines 6-8), followed by evaluation (lines 10-11). Note that describing a step via `setup` and `operation` systematically requires two inputs. The first is a description of the step, which is mandatory to associate a measure to an operation. The second is a *method reference* (e.g., `this::setData` refers to the `setData` method of the class implementing the protocol). Each method includes a compound block of instructions. As already said, the goal of this design pattern is to see a protocol as an array of method references. As a side effect, this results in code which is free from ad-hoc timers for measuring (as this code can be factorized).

We understand that this discussion mostly concerns engineering aspects of the tool, but we believe them to be critical to be able to share intelligible protocols. For space reasons, we refer to [1] for more details on B-Runner architecture.

4 Effective Experiment Conduction

We conclude by discussing two key aspects: outputs and limitations of the tool.

Interpreting Benchmark Results The goal of benchmarking is to yield data to be analyzed. B-Runner provides results in a structured format (XML or JSON) which suit routines for data aggregation. Monitoring test execution must also not be underestimated as tests can take a long time and/or fail. Benchmarking progress as well as errors are visible via logging and exported results [1].

Recognizing and Overcoming Limitations While B-Runner provides a flexible and extensible setting to automate testing scenarios, it also has some limitations. We will discuss how to circumvent some points that mainly pertain to the use of Java.

Monitoring Memory. This is a feature that B-Runner does not currently handle but which is planned for future releases. The strategy we pursue consists in using the JMH library facilities to plug a profiler for reporting custom metrics.

The API Wall. The measurement possibilities depend on the granularity of the methods exposed by the API of the tested system. B-Runner integrates seamlessly with Java APIs but it can still be used for tools implemented in other languages either via Java Native Interface (JNI) or system calls.

Comparing Versions. Comparing the performance of a tool to one of its previously published versions may raise a conflict of Java dependencies. To avoid this, two separate executions of the benchmarking activity are required (see [1]).

5 Related Work

Collaborative benchmarking has been little regarded for rule-based reasoning. The closest work to ours is [2] which provides an API and an implementation for testing Java reasoners inspired by the OpenRuleBench benchmark [11]. However, our system differs by proposing a more general and scalable architecture, which introduces a methodology for writing and sharing configurations and test protocols, as well as the use of JMH for controllable experiment conduction. Collaborative benchmarking is important in the context of *scientific workflows* [8]. These have been introduced for the reproducibility and automation of large scale scientific computations in domains such as genomics, biology, and astronomy [5]. These are expressed as a directed graph whose nodes are computations and edges dependencies. Research work in this area concerns the design of languages for workflow specification, as well as the reproducibility of experiments (or part of experiments) involving large data masses. The optimization of scheduling computations across distributed and cloud platforms has also been considered, with the goal to obtain results faster and with less cloud computation [13,12].

6 Conclusion and Outline

Principled benchmarking paradigms can be instrumental to perform robust experimental analysis at small cost. In particular, we advocate for the development of collaborative benchmarking for knowledge and rule-based reasoners. As an answer to this need, we introduced B-Runner, an open library for testing rule-based reasoners. B-Runner leverages on Java Microbenchmarking Harnessing (JMH) for experiment conduction. B-Runner is Java-based, which makes it portable, yet still able to support non-Java systems.

Our library implements an architecture for collaborative benchmarking which is service-oriented and configuration-driven. We argue that sharing reusable services dramatically reduces experiment setup thereby increasing their reproducibility. More generally, this provides testing accessibility to larger audiences. Also, making test protocols transparent through a simple yet well-defined design pattern enhances their robustness and scrutability. Crucially, this can contribute to increase the fairness of comparisons when these span across different tools. Using a configuration-based approach, as opposed to scripting, constitutes a simple way to plan extended benchmarking activities, which also improves readability. Finally, note that while B-Runner focuses on reasoning systems, the principles it implements can be applied to a larger extent.

B-Runner is available online [1]. The tool supports a number of reasoners and is currently under active development. Future work also involves the creation of a library for automatic chart creation from trial results.

Acknowledgements This work was financially supported by the ANR project CQFD (ANR-18-CE23-0003) and by the Inria-DFKI bilateral project R4Agri.

References

1. B-runner repository. `gitlab.inria.fr/rules/brunner` (2024)
2. Angele, K., Angele, J., Simsek, U., Fensel, D.: RUBEN: A rule engine benchmarking framework. In: Rule Challenge @ RuleML+RR 2022 (2022)
3. Baget, J.F., Bisquert, P., Leclère, M., Mugnier, M.L., Pérution-Kihli, G., Tornil, F., Ulliana, F.: InteGraal: a Tool for Data-Integration and Reasoning on Heterogeneous and Federated Sources. In: BDA 2023. Montpellier, France (Oct 2023)
4. Baget, J., Gutierrez, A., Leclère, M., Mugnier, M., Rocher, S., Sipieter, C.: Datalog+: Formats and translations for existential rules. In: RuleML (2015)
5. Barker, A., Van Hemert, J.: Scientific workflow: a survey and research directions. In: International Conference on Parallel Processing and Applied Mathematics. pp. 746–753. Springer (2007)
6. Beeri, C., Vardi, M.Y.: A proof procedure for data dependencies. J. ACM (1984)
7. Benedikt, M., Konstantinidis, G., Mecca, G., Motik, B., Papotti, P., Santoro, D., Tsamoura, E.: Benchmarking the chase. In: PODS (2017)
8. Cohen-Boulakia, S., Belhajjame, K., Collin, O., Chopard, J., Froidevaux, C., Gaignard, A., Hinsen, K., Larmande, P., Bras, Y.L., Lemoine, F., Mareuil, F., Ménager, H., Pradal, C., Blanchet, C.: Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities. Future Generation Computer Systems **75**, 284–298 (2017)
9. König, M., Leclère, M., Mugnier, M., Thomazo, M.: Sound, complete and minimal ucq-rewriting for existential rules. Semantic Web (2015)
10. Lenzerini, M.: Data integration: A theoretical perspective. PODS (2002)
11. Liang, S., Fodor, P., Wan, H., Kifer, M.: Openrulebench: an analysis of the performance of rule engines. In: WWW (2009)
12. Liew, C.S., Atkinson, M.P., Galea, M., Ang, T.F., Martin, P., Hemert, J.I.V.: Scientific workflows: moving across paradigms. ACM Computing Surveys (2016)
13. Liu, J., Lu, S., Che, D.: A survey of modern scientific workflow scheduling algorithms and systems in the era of big data. In: 2020 IEEE International Conference on Services Computing (SCC). pp. 132–141. IEEE (2020)
14. Manegold, S., Manolescu, I.: Performance evaluation in database research: principles and experience. In: EDBT (2009)
15. Mugnier, M., Thomazo, M.: An introduction to ontology-based query answering with existential rules. In: RR Summer School (2014)