PAPyA: a Library for Performance Analysis of SQL-based RDF Processing Systems

Mohamed Ragab\textsuperscript{a}, Adam Satria Adidarma.\textsuperscript{b,*} and Riccardo Tommasini\textsuperscript{c}

\textsuperscript{a} Computer Science Inst., University of Tartu, Tartu, Estonia
E-mail: mohamed.ragab@ut.ee
\textsuperscript{b} Sepuluh Nopember Institute of Technology, Surabaya, Indonesia
E-mail: adam.19051@mhs.tis.ac.id
\textsuperscript{c} LIRIS Lab INSA de Lyon, Villeurbanne, France
E-mail: riccardo.tommasini@ut.ee

Abstract. Prescriptive Performance Analysis (PPA) has shown to be more useful than traditional descriptive and diagnostic analyses for making sense of Big Data (BD) frameworks’ performance. In practice, when processing large (RDF) graphs on top of relational BD systems, several design decisions emerge and cannot be decided automatically, e.g., the choice of the schema, the partitioning technique, and the storage formats. PPA, and in particular ranking functions, helps enable actionable insights on performance data, leading practitioners to an easier choice of the best way to deploy BD frameworks, especially for graph processing. However, the amount of experimental work required to implement PPA is still huge. In this paper, we present PAPyA\textsuperscript{1}, a library for implementing PPA that allows (1) preparing RDF graphs data for a processing pipeline over relational BD systems, (2) enables automatic ranking of the performance in a user-defined solution space of experimental dimensions; (3) allows user-defined flexible extensions in terms of systems to test and ranking methods. We showcase PAPyA on a set of experiments based on the SparkSQL framework. PAPyA simplifies the performance analytics of BD systems for processing large (RDF) graphs. We provide PAPyA as a public open-source library under an MIT license that will be a catalyst for designing new research prescriptive analytical techniques for BD applications.

Keywords: Benchmarking, RDF Systems, Big Data, Apache Spark

1. Introduction

The increasing adoption of Knowledge Graphs (KGs) in industry and academia requires scalable systems for taming linked data at large volumes and velocity. In absence of a scalable native graph system for querying large (RDF) graphs [1], most approaches fall back to using relational Big Data (BD) frameworks (e.g., Apache Spark or Impala) for handling large graph query workloads [2, 3]. Despite its flexibility, the relational model requires several additional design decisions when used for processing graphs, which cannot be decided automatically, e.g., the choice of the schema, the partitioning techniques, and the storage formats.

In [4], we highlight the severity of the problem by showing the lack of performance replicability of BD systems for querying large (RDF) graphs. In particular, we showed that changing even just one experimental parameter, e.g., partitioning technique or storage encoding, invalidates existing optimizations in the relational representation of RDF data. We observe that issues do not lay in the way the investigations were conducted but rather in the maturity

\textsuperscript{*}Equal Contribution with the first author. Corresponding Author mail E-mail: mohamed.ragab@ut.ee.
\textsuperscript{1}https://github.com/DataSystemsGroupUT/PAPyA
of the analysis, which is limited to descriptive or at most diagnostic observations of the system behavior. Such discussions leave much work for practitioners to transform performance observations into actionable insights [5].

Later in [5], we have introduced the concept of Bench-Ranking as a means for enabling Prescriptive Performance Analysis (PPA) for processing larger RDF graphs. The PPA is an alternative to descriptive/diagnostic analyses that aims to answer the question What should we do? [6]. In practice, Bench-Ranking enables informed decision-making without neglecting the effectiveness of the performance analyses [5]. In particular, we showed how it could prescribe the best-performing combination of schema, partitioning technique, and storage format for querying large (RDF) graphs on top of SparkSQL framework [5].

Our direct experience with big RDF graphs processing shows that the most time-consuming phases were the data preparation and performance analytics. Although the Bench-Ranking methodology simplifies performance analyses, its current implementation is still limited as it does not follow any specific software engineering best practices. Thus, practitioners who implement the Bench-Ranking methodology may face the following challenges:

C.1 Experiment Preparation requires huge data engineering efforts to build the full pipeline for processing large graphs on top of BD systems, to put the data in the logical and physical representations that adapt with relational distributed environments. Moreover, the current experimental preparation in Bench-Ranking requires incorporating several systems, e.g., Apache Jena (i.e., for logical schema definitions), and Apache SparkSQL (i.e., for physical partitioning and storage).

C.2 Portability and Usability: deciding new requirements in the Bench-Ranking framework’s current implementation (e.g., changes over the number of tasks (i.e., queries) or changes over the experimental dimensions/configurations.) would lead to repeating vast parts of the work.

C.3 Flexibility and Extensibility: the current implementation of the Bench-Ranking framework does not fully reflect the flexibility and extensibility of the framework in terms of experimental dimensions and ranking criteria extensibility.

C.4 Complexity and Compoundness: practitioners may find Bench-Ranking criteria and evaluation metrics quite complex to implement. Moreover, the current implementation does not provide an interactive interface that eases interconnections of various modules of the framework (e.g., data and performance visualization).

To mitigate these problems, we design and implement an open-source library called PAPyA (Prescriptive Analysis in Python Actions), which contributes with (1) reducing the engineering work required for graph processing preparations and data loading; (2) reproducing existing experiments (according to user needs and convenience) for relational processing of SPARQL queries using SparkSQL. This will lead to reducing massive efforts for building analytical pipelines from scratch for relational BD systems that are subject to the experiments. Moreover, PAPyA also aims at (3) automating the Bench-Ranking methods for prescriptive performance analysis described in [5]. In practice, PAPyA facilitates navigating the complex solution space via packaging the functionality of different ranking functions as well as Multi-Optimization (MO) techniques into interactive programmatic library interfaces. Last but not least, (4) checking the replicability of the relational BD systems’ performance for querying large (RDF) graphs within a complex experimental solution space.

The focus of this paper is to show the internals and functionality of PAPyA as a means for providing PPA for BD relational systems that query large (RDF) graphs. For completeness, we aim to describe PAPyA prescriptions with the WatDiv benchmark [7] experiments 2. In [5], we applied Bench-Ranking to the SP2B benchmark [8]. However, the best-performing configurations depend on the dataset and the query workload [5]. Thanks to PAPyA, we can easily perform the PPA for any RDF benchmark, pointing out to the performance results, and specifying the experimental configurations (see Figure 1).

---

2Due to space limits, we diagnose the WatDiv prescriptions results and evaluation metrics in section 4 on the library GitHub page, mentioned below.
Outline. Section 2 presents the necessary preliminaries to understand the paper's content. Section 2.2 briefly introduces the Bench-Ranking framework concepts [5]. Section 3 presents the PAPyA requirements alongside its framework architecture. Section 4 shows how to use PAPyA in practice, providing examples from existing experiments [4, 5]. Section 5 discusses related work, while Section 6 concludes the paper and presents PAPyA's roadmap.

2. Background

This section presents the necessary background to understand the paper's content. We assume that the reader is familiar with the RDF data model and the SPARQL query language.

2.1. RDF Relational Processing Experimental Dimensions

Several design decisions emerge when utilizing relational BD systems for querying large (RDF) graphs, such as the relational schema, partitioning techniques, and storage formats. These experimental dimensions directly impact the performance of BD systems while querying large graphs. Intuitively, these dimensions entail different choice options (we call them dimensions' parameters).

First, the relational schema, it is easy to show that we have different options on how to represent graphs as relational tables, and the choice hugely impacts the performance [2, 9]. We identify the most used ones in the literature of RDF processing [2, 3, 9, 10]: Single Statement Table (ST) schema that stores RDF triples in a ternary-column relation (subject, predicate, object), and often requires many self-joins; Vertically Partitioned Tables (VP) proposed by Abadi et al. [9] to mitigate issues of self-joins in ST schema proposing to use binary relations (subject, object) for each unique predicate in the RDF dataset; the Property Tables (PT) schema that prescribes clustering multiple RDF properties as n-ary columns table for the same subject to group entities that are similar in structure. Lastly, two more RDF relational schema advancements also emerge in the literature. The Wide Property Table (WPT) schema encodes the entire dataset into a single denormalized table. WPT is initially proposed for Sempala system by Scätzle et al. [3], who also proposed another schema optimization that extended the version of the VP schema (ExtVP) [2] that pre-computes semi-join VP tables to reduce data shuffling.

BD platforms are designed to scale horizontally; thus, data partitioning is another crucial dimension for querying large graphs. However, choosing the right partitioning technique for RDF data is non-trivial. To this extent, we followed the indication of Akhter et al. [11]. In particular, we selected the three techniques that can be applied directly to RDF graphs while being mapped to a relational schema. Namely, (i) Horizontal Partitioning (HP) randomly divides data evenly on the number of machines in the cluster, i.e., \( n \) equally sized chunks, where \( n \) is the number of machines in the cluster. (ii) Subject-Based Partitioning (SBP) (or (iii) Predicate-Based Partitioning (PBP)) distributes triples to the various partitions according to the hash value computed for the subjects (predicates). As a result, all the triples with the same subject (predicate) reside on the same partition. Notably, both SBP and PBP may suffer from data skewness which impacts parallelism.

Serializing RDF data also offers many options such as RDF/XML, Turtle, JSON-LD, to name a few. On the same note, BD platforms offer many options for reading/writing to various file formats and storage backends. Therefore, we need to consider the variety of storage formats [12]. We specifically focus on the various Hadoop Distributed File System (HDFS) file formats that are suitable for distributed scalable setups. In particular, HDFS supports the following row-oriented formats (e.g., CSV and Avro) and columnar formats (e.g., ORC and Parquet).

2.2. Bench-Ranking in a Nutshell

This section summarizes the concept of Bench-Ranking as a means for Prescriptive Performance Analysis. Bench-Ranking is based on three fundamental notions, i.e., Configuration, Ranking Function, and Ranking Set, defined below.

Definition 1. A configuration \( c \) is a combination of experimental dimensions. The configuration space \( \mathcal{C} \) is the Cartesian product of the possible configurations.

Fig. 2.: The configuration space \( \mathcal{C} \).
In [5], we consider a three-dimensional configuration space, i.e., including relational schemas, partitioning techniques and storage formats. Figure 2 shows the experimental space and highlights the example of the (a.i.1) configuration, which is akin to the Single Triples (ST) schema, Subject-based Partitioning (SBP) technique, and stored in the HDFS (ORC) storage file format. This naming convention guides configurations reading in the rest of the paper results (figures and tables).

**Definition 2.** A ranking set \( R \) is an ordered set of elements ordered by a rank score. The rank index \( r_i \) is called the index of a ranked element \( i \) within the ranking set \( R \), i.e., \( R[r_i]=i \). We denote with \( R^k \) the leftmost (k top-ranked) subset of \( R \), and we denote with \( R_x \) the ranking set calculated according to the Rank score \( R_x \).

**Definition 3.** A ranking set is defined by a ranking function \( f_R : C \rightarrow R \) that associates a rank score to every element in \( C \).

A valid example of a ranking score can be the time required for query executions by each of the selected configurations (see Table 1). The ranking function abstracts this notion (Definition 3). In [5], we consider a generalized version of the ranking function presented in [11], which calculates the rank scores for the configurations as follows:

\[
R = \sum_{r=1}^{d} \frac{O_{\text{dom}}(r) \cdot (d - r)}{|Q| \cdot (d - 1)}, \quad 0 < R \leq 1
\]  

(1)

In Equation (1), \( R \) is the rank score of the ranked dimension (i.e., relational schema, partitioning technique, storage format, or any other experimental dimensions). Such that, \( d \) represents the total number of parameters (options) under that dimension (for instance five in case of schemas, see Figure 2), \( O_{\text{dom}}(r) \) denotes the number of occurrences of the dimension being placed at the rank \( r \) (1st, 2nd, ...). Moreover, \(|Q|\) represents the total number of queries. Rank scores define the Single-Dimensional (SD) ranking criteria that help to provide a high-level view of the system performance across a set of tasks (e.g., queries in a workload) [5]. Table 2 shows a simple example of applying the above formula for computing the rank scores of the relational schema dimension. The ExtVP schema was placed in the "first" rank (six times) (i.e., performed the best in six queries), the "second" (six times), the "third" (eight) times, the "fourth" (zero) times, and the last "fifth" also (zero) times. Thus, its overall ranking is 0.73. In contrast, the VP performed the worst with a rank score of 0.26. Intuitively, this means that the ExtVP schema in this example is the best (i.e., it has the highest rank score), and the VP schema is the worst-performing one.

Despite its generalization, Equation (1) is insufficient for ranking the configurations in a configuration set defined \( C \) when it counts multiple dimensions [5]. The presence of trade-offs [5, 13] reduce the accuracy of these SD ranking functions. Thus, we introduced Multi-Dimensional ranking [5] by approaching Bench-Ranking as a Multi-objective Optimization problem. In particular, we adopt the Pareto frontier optimization techniques \(^3\) implemented using Non-dominated Sorting Genetic Algorithm (NSGA-II) [14], to optimize the experimental dimensions altogether and find the best-performing configuration in \( C \).

Finally, our Bench-Ranking frameworks include two evaluation metrics to assess the effectiveness of the proposed ranking criteria. In particular, we consider a ranking criterion is good if it does not suggest low-performing configurations and if it minimizes the number of contradictions within an experimental setting. When it comes to PPA, \(^3\)Pareto frontier aims at finding a set of optimal solutions if no objective can be improved without sacrificing at least one other objective.
practitioners are not interested in a configuration being the fastest at answering any specific query in a workload as long as it is never the slowest at any of the queries. To this extent, we identified two evaluation metrics [5], i.e.,

(i) **Conformance**, which measures the adherence of the top-ranked configurations w.r.t. actual query results (see Table 1); (ii) **Coherence** which measures the level of (dis)agreement between two ranking sets that use the same ranking criterion across different experiments (e.g., different dataset sizes).

We calculate the conformance according to Equation (2) by positioning an element in a ranking set w.r.t. the initial rank score. For instance, let’s consider a ranking criterion R. A ranking criterion across different experiments (e.g., different dataset sizes).

Thus, the Conformance (R) k of the chosen KPI. In [5], we opted for query latency, yet one may need to analyze the performance in terms of other metrics (e.g., memory consumption).

\[
A(R^k) = 1 - \frac{1}{|Q|} \sum_{i=0}^{k} A(i, j) \quad \text{if} \quad \bar{R}[j] \in Q_{th}
\]

For coherence, we employ Kendall’s index \(^4\) according to Equation (3), which counts the number of pairwise (dis)agreements between two ranking sets, Kendall’s distance between two ranking sets \(R1\) and \(R2\), where \(P\) is the set of unique pairs of distinct elements. The larger the distance, the more dissimilar the ranking sets are.

\[
K_{i,j}(R1, R2) = \begin{cases} 
0 & \text{if } R1[r_i] = R2[r_j] = i \land R1[r_i] = j \land R2[r_j] = j \land r_i = r_j \\
1 & \text{otherwise}
\end{cases}
\]

K \((R1, R2) = \sum_{(i,j) \in P} K_{i,j}(R1, R2) \quad \text{if } \left| P \right|

We assume that ranking sets have the same number of elements. For instance, the K index between \(R1=\{a.ii.3, a.i.3, b.ii.2\}\) and \(R2=\{a.ii.3, a.i.3, c.iii.2\}\) for 100M and 500M is 0.33, i.e., one disagreement out of three comparisons.

3. **PAPyA**

In this section, we present the requirement analysis for PAPyA library, and describe its architecture. We elicit PAPyA’s requirements based on the implementation challenges we discussed in the introduction and on the existing research efforts on benchmarking BD systems for processing and querying large RDF graphs [2, 3, 5, 13, 15].

3.1. **Requirements**

R.1 **Support for PPA**: PAPyA shall support the necessary abstractions required to support PPA. Moreover, by default, it shall support existing Bench-Ranking techniques in [5].

R.2 **Independence from the Key Performance Indicators (KPIs)**: PAPyA must enable PPA independently from the chosen KPI. In [5], we opted for query latency, yet one may need to analyze the performance in terms of other metrics (e.g., memory consumption).

---

\(^4\)Kendall is a standard measure to compare the outcomes of ranking functions.
R.3 Independence from Experimental Dimensions: PAPyA must allow the definition of an arbitrary number of dimensions, i.e., allow the definition of n-dimensional configuration space.

R.4 Usability: PAPyA supports decision-making, simplifying the performance data analytics. To this extent, data visualization techniques and a simplified API are both of paramount importance.

R.5 Flexibility and Extensibility: PAPyA should be extensible both in terms of architecture and programming abstractions to adapt with adding/removing configurations, dimensions, workload queries, ranking methods, evaluation metrics, etc. It also should decouple data and processing abstractions to ease the integration of new components, tools, and techniques.

3.2. Architecture, Abstractions, and Internals

This section presents the PAPyA’s main components and shows how they fulfill the requirements. Table 3 summarizes the requirements to challenges mappings alongside the PAPyA solutions. PAPyA allows its user to build an entire pipeline for querying big RDF datasets and analyzing the performance results. In particular, it facilitates building the experimental setting considering the configuration space (described in Definition 1) specified by users. This entails preparing and loading the graph data in a user-defined relational configuration space, then performing experiments (executing a query workload on top of a relational BD framework), and finally analyzing and providing prescriptions of the performance.

To achieve that, PAPyA includes three core modules depicted in Figure 3, i.e., the Data Preparator, the Executor, and the Ranker. Moreover, PAPyA relies on few core abstractions depicted in Figure 4, i.e., Configuration, Experiment, Result, and Rank. While detailing each module’s functionalities, we introduce PAPyA workflow, which also appears in Figure 3, starting with the input is a configuration file that points to the input N-Triple file with the RDF graph (Figure 3 step (A)).

The first actor in the pipeline is the Data Preparator (DP), which prepares RDF graphs for relational processing. It takes as input a configuration file that includes experimental options of interest. The configuration file is represented by the Configuration abstraction (see Figure 4), which enables extensibility (R.5). Specifically, the DP allows defining an arbitrary number of dimensions with as many options as specified (R.3). In particular, it considers the dimensions specified in [5] (R.1), i.e., relational schemas, storage format, and partitioning technique. Therefore, the DP automatically prepares (i.e., determines, generates, and loads) the RDF graph dataset with the specified configurations that adapt with the relational processing paradigm to the storage destination (e.g., HDFS). More specifically,
DP currently includes four relational schemas commonly used for RDF processing, i.e., (a) ST, (b) VP, (d) ExtVP, and (e) WPT. For partitioning, DP currently supports three partitioning techniques, i.e., (i) horizontal partitioning (HP), (ii) subject-based partitioning (SBP), and predicate-based partitioning (PBP). Last but not least, DP enables storing data using four HDFS file formats (i) CSV or (ii) Avro, which are row-oriented, and (iii) ORC or (iv) Parquet, which are column-oriented storage formats [12]. The DP interface is generic, and the generated datasets are agnostic to the underlying relational system. The DP prepares RDF graph data for processing with different relational BD systems, especially SQL-on-Hadoop systems, e.g., SparkSQL, Hive, Impala. Seeking scalability, the current DP implementation relies on SparkSQL, which allows implementation of RDF relational schema generation using the SQL transformations. Notably, Apache Hive or Apache Impala could be potential candidates for an alternative implementation executors. However, SparkSQL also supports different partitioning techniques and multiple storage formats, making it ideal for our experiments [5].

Figure 5 shows sample of schema generation in PAPyA DP component. First, the DP transforms the input RDF graph (N-Triples file(s)) into an ST table schema (i.e., Figure 5 Step(1)), and then other schemas are generated using parameterized SQL queries. For instance, the VP and WPT schemas are generated using SQL queries given the ST table as a parameter (i.e., Figure 5 Step(2), and (3), respectively). While, the ExtVP schema generation relies on VP tables to exist first (i.e., Step(4) in Figure 5).

The Executor is the following module in PAPyA workflow, which is the system that is subject to experimentation (see Figure 3 step B). For instance, in [2, 4, 5, 15] the considered system is Apache SparkSQL. The executor offers an abstract API to be extended (R.5). In practice, it (i) starts the execution pipeline in the external system, (ii) it collects the performance logs (R.2), and currently, it persists them on a file system, e.g., HDFS. The Executor expects a set of experiments to run defined in terms of (i) a set of queries, (ii) an RDF dataset (size), and (iii) a configuration (defined in Definition 1). The Experiment abstraction is defined in Figure 4. We decide to wrap the running experiments in a SparkSQL-based executor (SparkExecutor). The experiment specifications (alongside the configurations) are passed to this wrapper as parameters. It is worth mentioning that the Executor assumes the query workload is available in the form of the SQL queries. In the current stage, PAPyA does not support SPARQL query translation nor SQL query mappings.

The results logs are then loaded by the Ranker component into python Dataframes to make them available for analysis (see Figure 3 step (C)). The Ranker reduces the time required to calculate the rankings, obtain useful data visualizations, and determine the best-performing configurations while checking the performance replicability. To fulfill R.2, i.e., the Ranker component operates over a log-based structure whose schema shall be specified by the user in the input configurations. Moreover, to simplify the usage and the extensibility (R.5), we decoupled the

---

5 Automating PT schema ("b" in Figure 2) generation is not yet supported by the current PAPyA DP
6 schema SQL-based transformations are kept in the DP module on PAPyA’s GitHub repository due to space limits.
performance analytics (e.g., ranking calculation) from its definitions and visualization. In particular, the Rank class abstraction reflects on the ranking function (Definition 3) that takes as input data elements and returns a ranking set. To fulfill R.4, a default data visualization for the rank shall be specified. However, this is left for the user to specify due to the specificity of the visualization.

The Rank call allows defining additional ranking criteria (R.5). In addition, to fulfill R.1, PAPyA already implements SD as well as Multi-Dimensional (MD) criteria specified in [5].

PAPyA allows its users to interact with the experimental environment (R.5) using Jupyter Notebook. To facilitate the analysis, it integrates data visualization (R.4) that can aid decision-making. Thus, the Rank class includes a specific method to implement, where to specify the default visualization.

Finally, to evaluate the ranking criteria, we introduced in Section 2.2 the notions of coherence and conformance (R.1). Ranking criteria evaluation metrics are employed to select which ranking criterion is “effective” (i.e., if it is not suggesting low-performing configurations). In our experiments, we use such metrics by looking at all ranking criteria and comparing them with the results across different scales, e.g., dataset sizes (100M, 250M, and 500M). Notably, to minimize the dependencies, we implemented the ranking algorithms and evaluation metrics from scratch.

### 4. PAPyA in Practice

In this section, we explain how to use PAPyA in practice, showcasing its functionalities with a focus on performance data analysis, flexibility, and visualizations. In particular, we design our experiments in terms of (i) a set of SPARQL queries that we manually translated into SQL accordingly with the different relational schemas, (ii) RDF datasets of different sizes automatically prepared using our Spark-based DataPreparator, and (iii) a configuration based on three dimensions as in [5], i.e., schema, partitioning techniques, and storage formats.

In Bench-Ranking experiments [5], we used the SP²B [8] benchmark datasets. In this paper, we aim to use a different benchmark to check the robustness of PAPyA Bench-Ranking criteria and their evaluation metrics. Thus, we present the results of

<table>
<thead>
<tr>
<th>D</th>
<th>WatDiv</th>
<th>WatDiv</th>
<th>Pareto</th>
<th>Pareto</th>
<th>Pareto</th>
</tr>
</thead>
<tbody>
<tr>
<td>100M</td>
<td>a.i.3</td>
<td>a.i.3</td>
<td>c.i.3</td>
<td>c.i.3</td>
<td>a.i.3</td>
</tr>
<tr>
<td></td>
<td>a.i.4</td>
<td>a.i.4</td>
<td>c.i.4</td>
<td>c.i.4</td>
<td>a.i.4</td>
</tr>
<tr>
<td></td>
<td>b.i.1</td>
<td>b.i.1</td>
<td>c.i.2</td>
<td>c.i.2</td>
<td>b.i.1</td>
</tr>
<tr>
<td></td>
<td>c.i.1</td>
<td>c.i.1</td>
<td>c.i.1</td>
<td>c.i.1</td>
<td>c.i.1</td>
</tr>
<tr>
<td>250M</td>
<td>a.i.3</td>
<td>a.i.3</td>
<td>c.a.4</td>
<td>c.a.4</td>
<td>a.i.3</td>
</tr>
<tr>
<td></td>
<td>a.i.4</td>
<td>a.i.4</td>
<td>c.a.4</td>
<td>c.a.4</td>
<td>a.i.4</td>
</tr>
<tr>
<td></td>
<td>c.a.2</td>
<td>c.a.2</td>
<td>c.a.2</td>
<td>c.a.2</td>
<td>c.a.2</td>
</tr>
<tr>
<td></td>
<td>c.a.1</td>
<td>c.a.1</td>
<td>c.a.1</td>
<td>c.a.1</td>
<td>c.a.1</td>
</tr>
<tr>
<td>500M</td>
<td>a.i.3</td>
<td>a.i.3</td>
<td>c.i.3</td>
<td>c.i.3</td>
<td>a.i.3</td>
</tr>
<tr>
<td></td>
<td>a.i.4</td>
<td>a.i.4</td>
<td>c.i.3</td>
<td>c.i.3</td>
<td>a.i.4</td>
</tr>
<tr>
<td></td>
<td>c.i.2</td>
<td>c.i.2</td>
<td>c.i.2</td>
<td>c.i.2</td>
<td>c.i.2</td>
</tr>
<tr>
<td></td>
<td>c.i.1</td>
<td>c.i.1</td>
<td>c.i.1</td>
<td>c.i.1</td>
<td>c.i.1</td>
</tr>
</tbody>
</table>

### Table 4
WatDiv best-performing (Top-3) configurations according to the SD and MD ranking criteria.

```python
from pappy import Configuration, SparkExecutor
from pappy import data_preparator as dp
#Configuration
conf = Configuration(
    "schemas": ["ST", "VP", "WPT", "ExtVP"],
    "partition": ["HP", "SBP", "PBP"],
    "storage": ["CSV", "Avro", "Parquet", "ORC"]
)
#Executor
Q = {ql1, ql2, ..., ql20}
exp = dp.experiment(data_set="100M", Q, confs)
spe = SparkExecutor(master=local[
    ])
res = spe.run(exp, runs=5, dataPath=dfs(...), logsPath=dfs(...))
#Benchmark-Ranking
#(1) SD-Ranking Criteria
schemaSDRanks = SDRank(res, dim=list(conf.keys())(0),
q=ql(Q), d=dim(list(conf.values())(0)), schemaSDRanks = ...
partitioningSDRanks = ...
storageSDRanks = ...
#MD-Ranking Criteria
poretoFronts_Q=MDRankPareto(dims=100M, Q, poretoFronts_Agg=MDRankPareto(dims=100M
, .dims=[schemaSDRanks, partitioningSDRanks, storageSDRanks]])
#Visualization
SDRank.plot(schemaSDRanks) #plot SDRanking for schema
MDRankPareto.plot(poretoFronts) #plot MD-Ranking Pareto
#Ranking Criteria evaluation
conf=Ranker.coherence(schedaSDRanks, q=20, k=3, b=45)
coh=Ranker.coherence(schemaSDRanks, 100M, schemaSDRanks, 250M)
```

Listing 1: Experiment design example in PAPyA.
experiments opting for the WatDiv benchmark [7]. WatDiv benchmark includes a data generator and a query workload with various graph patterns, SPARQL query shapes, and selectivities that make the analysis non-trivial. In our experiments, which are based on the average results of five runs, we measure the performance of SparkSQL as a BD relational engine in terms of query latency. However, alternative KPIs, e.g., memory consumption, could be easily used in PAPyA.

Listing 1 shows a full example of PAPyA pipeline, starting by deciding the configurations (in terms of three dimensions and their options, e.g., list of schemas, partitioning techniques, storage formats to prepare, load, and benchmark) (Listing 1 lines 4-8). Then, an experiment is set up for running, defining the dataset size (e.g., "100M" triples), a list of queries to execute or exclude from the workload, and the configurations (Listing 1 line 10). An executor is defined for running the experiment along with the number of times experiments will be run (Listing 1 line 11). The results (runtime logs) are kept in log files in a specified path (e.g., HDFS or a local disk). The BenchRanking phase starts when we have the results in logs (Listing 1 line 13). For instance, we call the SDRank (Listing 1 line 15) for calculating rank scores for the "schema" dimension, alongside specifying the number of queries ("q") and number of options under this dimension ("d" in Equation (1)). The MD-Ranking (i.e., Pareto fronts) is applied in two ways. The first one is called ParetoAgg (Listing 1 line 20), which applies the NSGA-II algorithm by considering the ranking sets obtained while sorting each query results individually. Using the first method, the algorithm aims at minimizing the query runtimes across all dimensions. The second one is called the ParetoAggMin (Listing 1 line 21), which operates on the SD ranking criteria. By using the second method, the algorithm aims to maximize the rank scores of the three SD-ranking criteria altogether, i.e., Rs, Rp, and Rf. The user can plot the SD rank scores and the MD Pareto ranking criterion (Listing 1 line 23).

Table 4 shows the top-3 ranked configuration according to the various ranking criteria, i.e., Single-Dimension and Multi-Dimensional (Pareto) for the WatDiv datasets (i.e., 100M, 250M, 500M triples). In addition, Table 5 provides the ranking evaluation metrics (calculated according to Equations (2) and (3)).

4.1. Rich Visualizations

To fulfill R4, PAPyA decouples data analytics from visualizations. Meaning that the user can specify his/her visualizations of interest with the performance data. Nevertheless, PAPyA yet provides several interactive and extensible default visualizations that help practitioners rationalize the performance results and final prescriptions. In addition, visualizations are simple and intuitive for understanding several Bench-Ranking definitions, equations, and evaluation metrics. For instance, Figure 6 (a-c) shows three samples of SD-ranking criteria plots. In particular, they show how many times a specific dimension’s (e.g., schema in Figure 6 (a)) alternatives/options (ST, VP, PT,...etc) achieve the highest or the lowest ranking scores. Figure 7 shows the SD ranking criteria w.r.t a simple geometrical representation (detailed in the following sections) that depicts the triangle subsumed by each dimension’s ranking criterion (i.e., Rs, Rp, and Rf). The triangle sides present the trade-offs ranking dimensions and show that the SD-

---

7 Nonetheless, seeking conciseness, we keep SP²B results on the GitHub repository.
8 Benchmarks’ query workload (in SQL) and experiments runtime: https://datasystemsgroup.github.io/SPARQLRDFBenchmarking
9 It is worth noting that the performance analyses, e.g., Bench-Ranking, could start directly if the performance data (logs) are already present.
Fig. 6. Examples on SD Rank Scores over different dimensions (100M), the higher the better.

Fig. 7. Dimensions trade-offs using single-dimensional ranking ($R_s, R_f$, and $R_p$).

Fig. 8. Pareto Fronts, and queries best-worst configuration examples.

Fig. 9. Heatmap shows the coherence of the $R_s$ criterion (Top-5 configurations) scaling from 100M to larger dataset scales. The stacked plot shows the Conformance of the top-3 ranked configurations.
ranking criteria may only optimize towards a single dimension at a time. The MD-ranking criteria, i.e., ParetoAgg\(^{10}\) results are depicted using a 3D plot in Figure 8 (a). Pareto fronts are depicted by the green shaded area of the three experimental dimensions of the Bench-Ranking (for WatDiv 500M triples dataset\(^{11}\)). Each point of this figure represents a solution of rank scores (i.e., a configuration in our case).

PAPyA visualizations allow explaining the conformance and coherence results using simple plots. For instance, Figure 9 (a) shows the coherence of the top-5 ranked configurations of the Rs criterion in the 100M dataset while scaling to the larger datasets, i.e., 250M and 500M. PAPyA explains the conformance of the Bench-Ranking criteria by visualizing the conformance of the top-3 ranked configurations (or any arbitrary number of configurations) with the actual query rankings (Table 1). The green color represents the level of conformance, and the red depicts a configuration is performing worse than the \(h\) worst rankings. Thus, this may explain why \(R_p\) and \(R_f\) criteria have low conformance results in Table 4, while the other criteria have relatively higher conformance values.

Practitioners can also use PAPyA visualizations for fine-grained ranking details. For instance, showing the best and worst configurations for each query (as shown in Figure 8 (b) for example of three queries of the WatDiv workload). Such detailed visualizations could help the user rationalize the final prescriptions of PAPyA.

### 4.2. PAPyA Flexibility & Extensibility

**Adding/Removing Configurations/Queries.** To show an example of the extensibility of PAPyA, we implement the Bench-Ranking criteria over a subset of the configurations and subset of the WatDiv benchmark tasks (i.e., queries); we call it WatDiv\(^{mini}\). In particular, we run PAPyA Bench-Ranking with WatDiv excluding two schemas (i.e., schema advancements: ExtVP, and WPT), one partitioning technique (i.e., Predicate-based), and one storage format (i.e., Avro). Then, we include all the configurations back to test the extensibility with the WatDiv experiments (see the left part of Table 4). The configurations’ exclusion and inclusion is specified easily from the YAML configuration file (as shown in Listing 2 lines 7-12), i.e., PAPyA considers only the specified configurations and rank according to them.

The right side of Table 4 shows the top-ranked three configurations according to the specified configurations. Intuitively, results differ according to the available ranked configuration space. For instance, with the inclusion of the ExtVP schema (i.e., ’d’), it dominates instead of the PT schema (i.e., ’c’) in WatDiv\(^{mini}\) for ranking by schema (Rs) criterion. In WatDiv\(^{mini}\), excluding the Predicate partitioning (’iii’), the subject-based partitioning (’ii’) completely dominates (one exception) in the \(R_p\) criterion across the different dataset sizes. Including it back, the predicate-based partitioning (’ii’*) significantly competes with the subject-based partitioning technique in most of the ranking criteria, i.e., \(R_p, R_f, \text{ParetoAgg}/Q, \text{and } R_{st}\).

With such flexibility, PAPyA also provides several dynamic views on the ranking criteria. For example, Table 6 shows the SD ranking of the schema dimension with changing the configuration space. Particularly, it shows how the global ranking of each relational schema (or any other specified dimension) could change by including/excluding configurations of the other dimensions. The table shows that the order of the global schema ranks changes by including all configurations (“Full Conf. Space”) than including/excluding the predicate partitioning or CSV format, due to space limits, we keep other Pareto figures on the PAPyA GitHub page.

---

\(^{10}\)Pareto\(_Q\) cannot be visualized, i.e., as it uses more than three dimensions, one for each query of the workload [5].

\(^{11}\)Due to space limits, we keep other Pareto figures on the PAPyA GitHub page.
i.e., "PBP/PBP", "CSV/CSV", respectively. For instance, the PT schema global ranking is interestingly oscillating with those changes in the available configurations.

<table>
<thead>
<tr>
<th>( D_i )</th>
<th>( R_i )</th>
<th>( R_j )</th>
<th>Pareto(_{\max} )</th>
<th>Pareto(_{max} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>100M</td>
<td>d1</td>
<td>a1</td>
<td>c1</td>
<td>c1</td>
</tr>
<tr>
<td>c1</td>
<td>d1</td>
<td>b1</td>
<td>b1</td>
<td></td>
</tr>
<tr>
<td>c1</td>
<td>d1</td>
<td>e1</td>
<td>e1</td>
<td></td>
</tr>
<tr>
<td>250M</td>
<td>c1</td>
<td>a1</td>
<td>d1</td>
<td>c1</td>
</tr>
<tr>
<td>d1</td>
<td>e1</td>
<td>d1</td>
<td>d1</td>
<td></td>
</tr>
<tr>
<td>c1</td>
<td>d1</td>
<td>c1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c1</td>
<td>c1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>500M</td>
<td>c1</td>
<td>a1</td>
<td>d1</td>
<td>c1</td>
</tr>
<tr>
<td>c1</td>
<td>e1</td>
<td>c1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7

Best-performing configurations, excluding the partitioning dimension.

Adding/Removing Full Experimental Dimension. PAPyA’s flexibility extends to the experimental dimensions, i.e., it is possible to add/remove dimensions easily. For instance, we can fully exclude the partitioning dimension in case experiments are executed on a single machine (see Listing 2 lines 13-18). In [16], we run experiments on SparkSQL with different relational schemas and storage backends yet without data partitioning. Table 7 shows the best-performing (top-3) configurations in WatDiv experiments when excluding the partitioning dimension. Table 8 shows the conformance and coherence metrics’ results for the various ranking criteria

<table>
<thead>
<tr>
<th>Metric</th>
<th>( D )</th>
<th>( R_i )</th>
<th>( R_j )</th>
<th>Pareto(_{\max} )</th>
<th>Pareto(_{max} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conform.</td>
<td>100M</td>
<td>95.00%</td>
<td>67.50%</td>
<td>95.00%</td>
<td>92.50%</td>
</tr>
<tr>
<td></td>
<td>250M</td>
<td>95.00%</td>
<td>30.00%</td>
<td>100.00%</td>
<td>97.50%</td>
</tr>
<tr>
<td></td>
<td>500M</td>
<td>100.00%</td>
<td>52.50%</td>
<td>100.00%</td>
<td>52.50%</td>
</tr>
<tr>
<td>Coher.</td>
<td>100M-250M</td>
<td>0.11</td>
<td>0.17</td>
<td>0.11</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>100M-500M</td>
<td>0.28</td>
<td>0.16</td>
<td>0.30</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>250M-500M</td>
<td>0.17</td>
<td>0.08</td>
<td>0.15</td>
<td>0.09</td>
</tr>
</tbody>
</table>

Table 8

Criteria evaluation (conformance and coherence), excluding partitioning.

Adding Ranking Criterion. PAPyA abstractions enable users to plug-in new ranking criteria besides the already existing ones (i.e., the abstract Rank class, Section 3). Let’s assume we seek usage of a simple ranking criterion that leverages a geometric interpretation of the SD rankings of the three experimental dimensions based on the triangle area subsumed by each ranking criterion (\( R_i, R_p, \) and \( R_J \)).

In Figure 10, the triangle sides represent the SD-ranking dimensions’ rank scores. Thus, this criterion aims to maximize this triangle’s area (i.e., the blue triangle). The closer to the ideal (outer red triangle), the better it scores. In other words, the bigger the area of this triangle covers, the better the performance of the three ranking dimensions altogether. The red triangle represents the case with the maximum/ideal rank score, i.e., \( R = 1 \) for the three dimensions (as, \( 0 < R \leq 1 \)). Equation (1) defines the blue triangle area; we call it ranking by triangle area (\( R_{\text{area}} \)).

\[
\text{TriangleArea}(R_{\text{area}}) = \frac{1}{2} \sin(120) \times (R_i \times R_p + R_i \times R_J + R_J \times R_i)
\]

The formula (Cf. Equation (4)) computes the actual triangle area. Simply, it sums up the triangle area of the three triangles \( A, B, \) and \( C \) by two of its sides which are the rank scores of each dimension, i.e., \( R_i, R_p, \) or \( R_J \) (dashed triangle sides), and the angle between both of them (i.e 120 in this case). For example, the actual area of the blue

Listing 3: MD-Ranking based on Pareto Fronts.

```
from papya import Rank

# Add Triangle_Area as a new ranking criterion.
class RtaCriterion(Rank):
    def calculate_rta(self):
        r_scores = SDRank(...).calculateRank()
        rta_scores = [0]
        for i in range(len(r_scores)):
            rs = r_scores[i]
            rp = r_scores[i+1]
            rf = r_scores[2+i]
        rta = (rs * rp + rs * rf + rp * rf) / 2
        return rta_scores

def plot(self): # plots (Figure 7)
    ...
```

12Notably, the Rp and Rta criteria cannot be calculated when excluding the partitioning dimension.
Fig. 11. Schema Replicability across changing partitioning or storage formats.

triangle is $R_{ta} = \frac{1}{2} \sin(120)(0.75 \cdot 0.771 + 0.73 \cdot 0.77 + 0.75 \cdot 0.73) = 0.73$. In addition to the SD and MD ranking criteria classes, Listing 3 shows how to extend PAPyA with a new Ranker class, i.e., $RtaCriterion$.

It is worth mentioning that the idea behind $R_{ta}$ is intuitively similar to ParetoAgg because both aim to maximize the rank scores of the three dimensions altogether. However, unlike ParetoAgg that is multi-dimensional, $R_{ta}$ cannot extend to dimensions above three. For simplicity, we used $R_{ta}$ as an exemplar to showcase that PAPyA abstractions enable extending new ranking algorithms/criteria. Table 4 shows the top-5 best-performing configurations ranked by $R_{ta}$. Results show that ParetoAgg results perfectly conform with $R_{ta}$ top-ranked configurations (especially in the top-3 ranked configurations). Table 5 also show that $R_{ta}$ criterion scores high conformance ratios across WatDiv benchmark datasets. It also scores high coherence (few disagreements) through the scalability of WatDiv datasets. In both WatDiv experiments (i.e., with mini and full dimensions inclusion), the $R_{ta}$ conformance and coherence values are very close to the ParetoAgg criterion.

4.3. Checking Performance Replicability

PAPyA also activates the functionality of checking the BD system’s performance replicability when introducing different experimental dimensions. In particular, it enables checking the system’s performance with one specific dimension while changing the parameters of the other dimensions. For example, Figures 11 (a) and (b) respectively show the impact of the partitioning and storage on the performance of the schema dimension. The Figures show how the performance of the system with a configuration can significantly change with changing other dimensions.

PAPyA can also check the performance replicability by comparing two configurations as discussed in [4]. For instance, PAPyA can compare the schema optimizations (i.e., WPT, and ExtVP) w.r.t their baseline ones (i.e., PT, and VP) while introducing different partitioning techniques and various HDFS storage formats that are different from the baseline configurations [2, 3].

Table 9 shows the effect of introducing partitioning techniques (right of the table) and different file formats (left of the table) different from the baseline configurations (i.e., Vanilla HDFS partitioning, and Parquet as storage format). The trade-offs effect is clear on the replicability results. Indeed, WPT outperforms PT schema only with 54.16% in the queries using the baseline Vanilla HDFS partitioning technique across all storage formats and only about 39% for the baseline Parquet format across all partitioning techniques. On the other side, we observe significant degradation of WPT schema optimization moving to other configurations with both partitioning and storage dimensions. For instance, WPT outperforms PT only with about 8% and 7% using other different partitioning techniques, i.e., Horizontal and Subject, respectively. The same occurs with changing the storage formats different from baseline Parquet. Similarly, ExtVP versus VP schema performance results con-
firm our observations. PAPyA enables showing those trade-offs of considering alternative storage file formats and partitioning techniques alongside the experiments’ query evaluation.

5. Related Work

The Semantic Web community is synergizing efforts pushing technological progress through benchmarking initiatives like the Linked Data Benchmark Council (LDBC)\(^{13}\), and providing full-fledge benchmarks like SP²Bench [8], WatDiv [7] to name few. In the literature, several tools and frameworks seek to reduce the effort required to design and execute reproducible experiments, share their results, and build pipelines for various Semantic Web applications. For instance, the RSPLab provides a test-drive for stream reasoning engines that can be deployed on the cloud. It enables the design of experiments via a programmatic interface that allows deploying the environment, running experiments, measuring the performance, and visualizing the results. In the work in [17], the authors tackled the problem of quantifying the continuous behavior of a query engine and presented two novel experimental metrics that capture the performance efficiency during an elapsed period rather than a constant instance of time. On another side, authors in [18] provide a data-loader that facilitates generating RDF graphs in different logical partitioning schemas and physical partitioning options. However, this tool stops the work of data generation and data loading. That is, this tool leaves the work of deciding the best solutions for the data/knowledge engineers.

The mentioned efforts aim at providing the environment that enables the practitioners to develop their experimental pipelines. Nonetheless, none of these efforts provide prescriptive performance analyses in the context of BD problems. Conversely, PAPyA aims to cover this timely research gap in an easy and extensible approach, facilitating building the experimental solution space, and automating providing prescriptions over it.

6. Conclusion and Roadmap

This paper presents PAPyA, an extensible library that reduces the efforts needed to analyze the performance of BD systems used for processing large (RDF) graphs. PAPyA implements the performance analytics methods adopted in [2, 3, 16] including a novel approach for prescriptive performance analytics we presented in [5].

Inspired by Gartner’s analysis methodology [19], Figure 12 reflects the amount of human intervention required to make a decision with the descriptive and diagnostic analyses of the performance results. Descriptive and diagnostic analytics are limited, and cannot guide practitioners directly to the best-performing configurations in a complex solution space. This is shown in this paper with the lack of performance replicability (shown Section 4.3). Indeed, the performance of the BD system is affected by changing the configurations, e.g., oscillating schema performance with changing partitioning, and storage options (Figure 11). On the other side, PAPyA reduces the amount of work required to interpret performance data. It adopts the Bench-ranking methodology with which practitioners can easily decide the best-performing configurations given an experimental solution space with an arbitrary number of dimensions. Although, descriptive discussions are limited, PAPyA still provides several descriptive analytics and visualizations on the performance to explain the final decisions given by PAPyA. PAPyA also aims to reduce the engineering work required for building an analytical pipeline for processing large RDF graphs. In particular, PAPyA prepares, generates, and loads data ready for big relational RDF graph analytics.

PAPyA is developed considering the ease of use and the flexibility aspects allowing extending the library with an additional arbitrary number of experimental dimensions to the solution space. Moreover, PAPyA provides ab-

\(^{13}\)ldbcouncil.org
stration on the level of ranking criteria, meaning that the user can use his/her ranking functions for ranking the solution space. Seeking availability, we provide PAPyA as an open-source library under MIT license and published at a persistent URI. PAPyA’s GitHub repository includes tutorials and documentation on how to use the library.

As a maintenance plan, PAPyA’s roadmap includes:

1. covers the phase of query evaluation into PAPyA pipeline. In particular, we plan to provide native support of SPARQL via incorporating native triple stores for query evaluation.
2. incorporate SPARQL into SQL translation using advancements of R2RML mapping tools (e.g., OnTop) [20].
3. wraps other SQL-on-Hadoop executors to PAPyA; thus, the performance of the engines could also be compared.
4. use orchestration tools (such as Apache Airflow) to monitor the PAPyA pipelines.
5. integrate PAPyA with other tools like gmark [21] that generates graphs and workloads, making them ready for distributed relational setups.

Acknowledgments. We acknowledge support from the European Social Fund via IT Academy programme and the European Regional Development Funds via the Mobilitas Plus programme (grant MOBTT75).

References