Smart SPARQL Advisor: Guiding Users in Query Formulation with Performance Prediction

Abiram Mohanaraj
Aalborg University
abiramm@cs.aau.dk

Matteo Lissandrini
University of Verona
matteo.lissandrini@univr.it

Katja Hose
TU Wien
katja.hose@tuwien.ac.at

ABSTRACT

Writing SPARQL queries is often an iterative process, where users refine queries until they meet their information needs. However, long-running query executions can lead to inefficient workflows, as users must wait idly for results — potentially without success due to strict timeouts imposed by public endpoints. In this demo, we present the *Smart SPARQL Advisor* (SSA), a system that integrates Query Performance Prediction (QPP) to proactively mitigate these issues. By predicting query runtimes prior to execution, SSA alerts users to potentially slow or timeout-prone queries and, when necessary, employs a large language model (LLM) guided by latent representations from the QPP model to suggest alternative query formulations. We demonstrate that SSA enables users to identify performant queries and understand performance bottlenecks, thereby reducing idle time and avoiding unproductive query executions. Through this approach, SSA fosters more responsive and resource-efficient interactions with triplestores, enhancing both user experience and triplestore utilization.

PVLDB Reference Format:
Abiram Mohanaraj, Matteo Lissandrini, and Katja Hose. Smart SPARQL Advisor: Guiding Users in Query Formulation with Performance Prediction. PVLDB, 18(12): 5295 - 5298, 2025.
doi:10.14778/3750601.3750655

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dkw-aau/SSA.git.

1 INTRODUCTION

Formulating SPARQL queries over Knowledge Graphs (KGs) is often an iterative and time-consuming process. Users typically begin with a simple and generic query and progressively refine it by modifying graph patterns until the desired results are obtained [3], as illustrated in Figure 1. While existing tools support this process by suggesting semantically relevant formulations, they overlook a critical aspect: *long query execution times.*

In public triplestores, such as those hosting DBpedia[1] and Wikidata[2], where strict query timeouts are enforced, users may wait for minutes — only for the query to time out or return uninformative results after a long delay, requiring further reformulation. This disrupts the interactive nature of query formulation, leading to inefficient workflows, wasted computation, and reduced productivity.

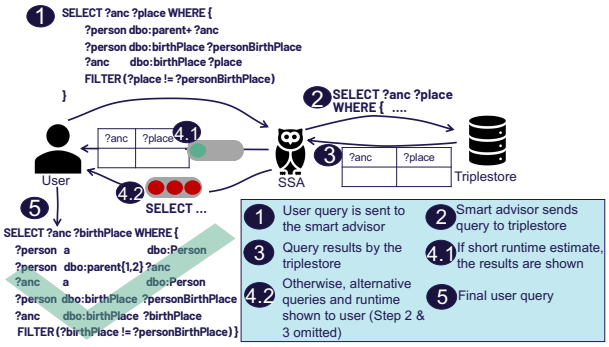

Figure 1: User iteratively changing the query

At the core of this challenge lies the structure of SPARQL and KGs themselves. SPARQL [4] is the standard for querying KGs in RDF, which are widely used for storing and querying large-scale heterogeneous graphs [4]. A KG is a set of subject-predicate-object triples [8]. As depicted in Figure 1, a core SPARQL operator is the Basic Graph Pattern (BGP), comprising a set of triple patterns — triples where the positions may contain variables. For instance, the subject and object in ⟨?s, dbo:parent, ?o⟩ in Figure 1 are variables that can match multiple nodes in the KG. *Despite the simple data model, writing SPARQL queries requires a deep understanding of the KG and is often non-trivial.*

Consequently, many systems have been proposed to support users during query formulation, including natural language interfaces [10], auto-completion [2], and query recommendation [6]. While these tools assist users in composing syntactically and semantically correct queries, they do not consider runtime characteristics during query authoring. As previously mentioned, since query formulation is inherently iterative, any significant delay caused by query evaluation leaves the user idle and interrupts the development workflow. *This motivates the integration of existing query authoring systems with functionalities that account for query runtime — a particularly valuable feature in public endpoints where strict time limits (typically a few minutes[1,2]) are enforced.*

In this paper, we propose *Smart SPARQL Advisor (SSA)*, owl in Figure 1. *SSA reduces idle time during query formulation by predicting slow queries and proactively suggests alternative similar queries.* SSA integrates two core components: 1) PlanRGCN [7] for Query Performance Prediction (QPP) to determine long-running queries and 2) a query recommendation method using an LLM with PlanRGCN to propose alternative fast queries to the user.

QPP is often used for learned query and workload optimization in DBMS [7, 9]. We demonstrate that it can also be used to guide users

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750655

[1]https://www.dbpedia.org/resources/sparql/
[2]Wikidata: https://www.mediawiki.org/wiki/Wikidata_Query_Service/User_Manual

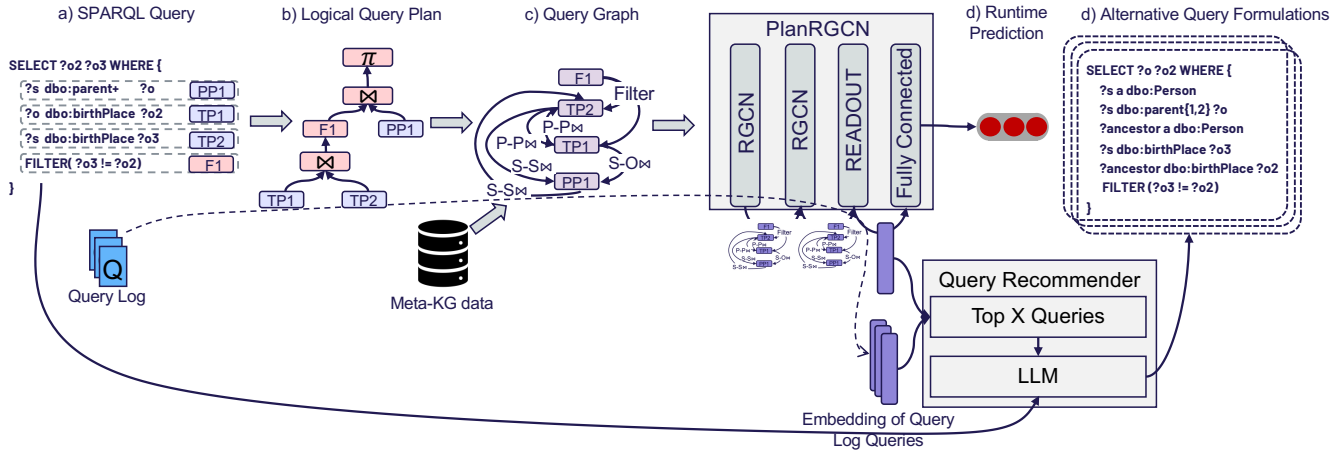

Figure 2: SSA Pipeline with Query Performance Prediction and Query Recommendation

towards query reformulations that require less time to complete, thus improving the query formulation process. If QPP predicts the user query to be fast, then our system will execute it in the triplestore and return the results to the user (Steps 1-4.1 in Figure 1). On the other hand, if the query is predicted to be slow, SSA will warn the user and recommend alternative queries (Steps 1,4.2 in Figure 1). Our recommendation method works by extracting similar fast-running queries using embeddings from PlanRGCN to prompt the LLM to revise the query using the extracted queries and the user query. In this demo, we showcase the following key contributions:

- A SPARQL Query Advisor (SSA) that determines if a user query is slow, and preemptively guides the user towards formulating a query with a shorter runtime.
- A query recommendation method using a novel, runtime-aware query representation and LLM-based query reformulation.

SSA is presented in Section 2, followed by a walkthrough of our demonstration scenario in Section 3, and we conclude by discussing the opportunities this offers in Section 4.

2 SMART SPARQL QUERY ADVISOR

SSA consists of a Web query editor and two components (i) a QPP component using PlanRGCN, which predicts query runtime and creates an embedding of the query, and (ii) a Query Recommender that uses QPP and an LLM to recommend alternative queries to the user. The pipeline is shown in Figure 2.

2.1 Query Performance Predictor

Performance prediction with PlanRGCN [7] works in three steps: 1) query plan representation, 2) query graph construction and featurization, and 3) Relational Graph Convolutional Network (RGCN). Note that query graphs do not correspond to execution plans, but instead represent generalized logical query plans used by PlanRGCN internally. In the first step, PlanRGCN transforms the unoptimized logical query plan of a SPARQL query into a *query graph* [7]. The query plan is a DAG, where nodes are logical operators, and edges denote data dependencies. From this, a query graph is constructed as a labeled multigraph where nodes correspond to operator types such as triple patterns (TPs), property paths (PPs), and FILTER expressions. Edges in the graph capture semantic relationships between operators, such as join connections (e.g., subject-subject, subject-object), OPTIONAL dependencies, and variable sharing with FILTERs. Join edges are labeled based on the variable positions involved in the connection (e.g., S-O, S-S). This graph construction ensures that logically equivalent queries yield identical query graphs by being operator-order agnostic, enabling PlanRGCN to learn over query structures, independent of query plan enumeration. A query graph example is depicted in Figure 2.c.

In the second step, each query-graph node is featurized with performance-relevant attributes derived from the target KG, which can easily be extracted via SPARQL queries [7]. Finally, an RGCN model learns query representations by aggregating information across the operator nodes and edge types of the featurized query graph. The output is a runtime interval prediction, i.e., the range where query execution is expected to finish.

PlanRGCN can be configured to predict runtime intervals across an arbitrary number of intervals, making the definition of a "long" query fully flexible and system-configurable. In practice, thresholds are often derived from the runtime distribution of historical query logs. For example, the 50th percentile separates fast from medium queries, while the 95th percentile distinguishes slow or timeout-prone queries [7]. In our demonstration, we train PlanRGCN to predict three runtime intervals, but for SSA's decision-making, we merge the first two and treat only the third as slow.

PlanRGCN offers three critical advantages over similar QPP approaches. First, PlanRGCN is triplestore-agnostic. Since SSA functions as a middleware between the user and a triplestore, we require a QPP method that does not depend on the internals or specific features of any one system. PlanRGCN satisfies this need by relying exclusively on logical plans and KG statistics that can be gathered through standard SPARQL queries or at loading time. This enables out-of-the-box deployment across various triplestores.

Second, PlanRGCN operates solely on unoptimized logical query plans, avoiding any dependency on physical query plans or optimizer decisions. In our context, extracting physical plans would require additional (and often engine-specific) interactions with the triplestore's internal APIs, undermining the lightweight nature of SSA. By using logical plans directly, PlanRGCN enables fast and

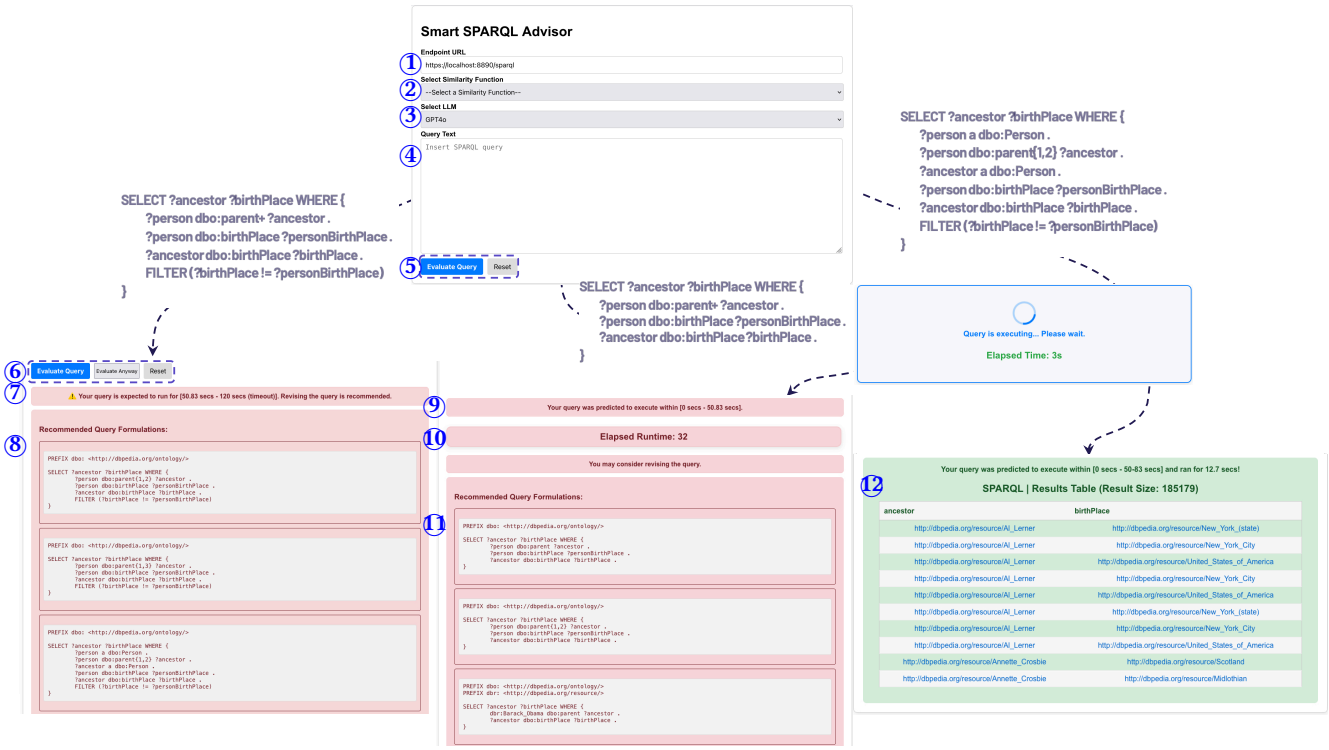

Figure 3: SSA demonstration scenario: the left arrow shows when SSA predicts the user query as slow, the middle arrow shows when the query is running potentially longer, and the right arrow shows fast query execution.

efficient runtime prediction without incurring overhead or tight coupling with the database engine.

Third, PlanRGCN can predict the performance of complex queries, even when predicates or entities were unseen at training time. PlanRGCN, and therefore SSA, currently supports BGPs, property paths, OPTIONAL, and FILTERs. SSA is ready to support a broader range of queries when they will be supported by PlanRGCN.

2.2 Query Recommendation via QPP and LLM

Upon detecting a long-running SPARQL query, we generate alternative query formulations by leveraging a PlanRGCN model and an LLM. First, we use PlanRGCN to obtain a latent vector representation $\mathbf{v}_Q$ of the input query's plan. Similarly, we compute latent representations $\mathbf{v}_{Q_i}$ for all queries Q_i in a query log $\mathcal{L}$, with runtime shorter than a given target running time τ (here we use the 50th percentile of query runtimes in $\mathcal{L}$). Next, we measure the similarity between the input query and each query in the query log using a vector similarity function $sim(\cdot,\cdot)$, such as the dot product or cosine similarity. Using these similarities, we retrieve the top X most similar queries: $\mathcal{S}=\{Q_i | S_i$ is among the top $X \wedge$ runtime$(Q_i)<\tau\}$

Finally, we inject a *prompt template* designed to guide the LLM towards generating runtime-efficient alternatives with the retrieved queries ($\mathcal{S}$) and the user query to prompt an LLM. The prompt templates are available in our GitHub repository. The LLM (e.g., GPT-4o) is prompted to generate N alternative queries. The intuition behind this approach is that the latent representations learned by PlanRGCN capture both the logical plan and the performance characteristics, allowing SSA to identify similar queries with fast executions. By leveraging these representations, we guide the LLM towards generating alternative query formulations that execute faster.

3 DEMONSTRATION SCENARIO

We demonstrate SSA on a snapshot of DBpedia loaded into Virtuoso[3]. In our demo, PlanRGCN has been configured to predict query runtime in the intervals $(0; 0.04]$, $(0.04; 50.8]$, or $(50.8; \infty]$ seconds. We consider queries with performance in $(0; 0.04]$ and $(0.04; 50.8]$ as fast-running queries, and $(50.8; \infty]$ as slow-running queries, similar to public endpoint timeouts.

Figure 3 illustrates the interactive web-based interface that users will engage with during the demonstration. The main screen comprises four key components (①–⑤):

- ①: A text field where users can specify the SPARQL endpoint URL, where the triplestore is available. Note that as a prerequisite for our system to work on the endpoint, we need a trained PlanRGCN model and statistics on the stored KG to be queried.
- ② & ③: Configuration options for query recommendation, including similarity functions ($sim(\cdot,\cdot)$) and LLM choice.
- ④: A query editor for users to input their SPARQL queries.

[3] http://virtuoso.openlinksw.com

- ⑤: Action buttons — "Reset" clears the input in ③, and "Evaluate Query" initiates the runtime prediction and query processing pipeline depending on the prediction.

Based on the PlanRGCN prediction and actual runtime, three scenarios are possible:

Scenario 1: Slow Query Prediction (Left Arrow). If PlanRGCN predicts the query to be slow-running, additional interface elements ⑦–⑧ are displayed, and ⑤ is replaced with ⑥.

⑥ is an updated action panel with an additional button allowing the user to proceed with evaluating the original query, despite PlanRGCN's slow runtime prediction. ⑦ presents the user with the runtime prediction information. ⑧ contains the list of alternative query formulations generated using LLM-guided recommendations based on PlanRGCN embeddings. Selecting one query formulation replaces the text in ④ and resets the interface to ①-⑤, enabling the user to proceed with the selected query.

Scenario 2: Potential Timeout (Middle Arrow). When a query is predicted to be fast but risks becoming slow (e.g., due to misprediction or workload/endpoint conditions), SSA enters a monitoring state. While the query is executing, a loading box is temporarily displayed below ⑤. If query runtime exceeds half of the threshold between fast and slow runtime intervals, runtime information about the prediction (⑨) and currently elapsed time (⑩) is displayed. Furthermore, we also provide the user with alternative queries in case the user can decide on a different formulation (⑪).

Scenario 3: Fast Execution (Right Arrow). The last scenario, depicted by the right arrow of the top page, describes when query performance is fast and predicted fast, where a loading box may briefly appear with the elapsed time information, similar to *Scenario 2*. Upon retrieving the query results, the loading box is replaced with ⑫ that displays the results with predicted and actual runtimes.

Results of QPP effectiveness. The effectiveness of SSA hinges on the prediction quality of PlanRGCN for QPP. For instance, if PlanRGCN often mispredicts slow queries as fast, then SSA's ability to reduce user idle time will be severely hindered. To evaluate this, we compare PlanRGCN against two state-of-the-art QPP methods: a neural network-based model [1] and an SVM-based approach [5]. We use a Dell R6415 server with a 16-core AMD 7281 CPU and 256 GB RAM, and a TITAN RTX GPU (24 GB, TU102) as our GPU. Extended experiments on PlanRGCN for QPP are available [7].

Table 1 reports the confusion matrix for all three models on our DBpedia test set, consisting of 3,895 queries and a KG of 6.1×10^9 triples. PlanRGCN processed an 11 k-query log in 83 min (feature extraction) and 64 min (training) [7]. Our results show that **PlanRGCN outperforms the baselines, especially in identifying $(50.8; \infty]$ queries**. In this interval, PlanRGCN achieves 78.2% accuracy, while the baselines misclassify a significant portion of slow queries as faster. Notably, the NN-based method misclassifies nearly 80% of $(50.8; \infty]$ queries, which would result in significant user idle time for the user. In contrast, PlanRGCN provides much more reliable performance on $(50.8; \infty]$ queries. For I2, PlanRGCN has reasonable performance, although 33.9% of the queries are misclassified as I3. While this is not ideal, such mispredictions are handled in our system through *Scenario 2*, which monitors the query executions and prompts the user to reconsider the query.

Table 1: Confusion matrix on runtime intervals on DBpedia

	P			NN			SVM			# Total
	I1	I2	I3	I1	I2	I3	I1	I2	I3	
I1: (0; 0.04]	85.2	12.6	2.2	**89.3**	10.6	0.1	0.6	99.4	0.0	1788
I2: (0.04; 50.8]	13.3	51.4	35.3	12.9	**85.7**	1.4	0.3	99.7	0.0	1688
I3: (50.8; ∞]	1.9	19.9	**78.2**	1.4	79.0	19.6	0.0	100.0	0.0	419

Altogether, these results confirm that *PlanRGCN strikes a better balance across runtime intervals, making it a reliable and practical choice for QPP in SSA*. Its strong predictive performance on the *slow* interval ensures that users are effectively warned about long waiting times, while still maintaining a good accuracy for fast queries – enabling more responsible and productive query formulation workflows. QPP inference takes on average 0.013 seconds while LLM-based reformulation takes ≈3.33 seconds (Phi-3). This overhead is acceptable since the alternative would be to wait for the query to time out (ca. 1-2 minutes).

4 CONCLUSION

In this paper, we presented Smart SPARQL Advisor (SSA), an interactive system that combines query performance prediction and LLM-based recommendation to guide users in crafting efficient SPARQL queries. We demonstrate the opportunities offered by proactively predicting query runtimes (using PlanRGCN) and considering query runtime in the query authoring process. SSA reduces user idle time, avoids wasted server resources, and supports a smoother query formulation experience. SSA advocates for observing how runtime-aware guidance and intelligent query recommendations can accelerate SPARQL development and improve responsiveness in triplestore interactions. Further, SSA showcases a controlled use of LLMs as a query reformulator by exploiting an accurate selection of similar relevant queries.

ACKNOWLEDGMENTS

This research was partially funded by the Independent Research Fund Denmark (DFF) under grant agreement no. DFF-8048-00051B and the Poul Due Jensens Fond (Grundfos Foundation).

REFERENCES

[1] Daniel Arturo Casal Amat, Carlos Buil Aranda, and Carlos Valle-Vidal. 2021. A Neural Networks Approach to SPARQL Query Performance Prediction. In *CLEI*.
[2] Hannah Bast, Johannes Kalmbach, Theresa Klumpp, Florian Kramer, and Niklas Schnelle. 2022. Efficient and Effective SPARQL Autocompletion on Very Large Knowledge Graphs. In *CIKM'22*. 2893–2902.
[3] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *VLDB J.* 29, 2-3 (2020), 655–679.
[4] Steve Harris and Seaborne Andy. 2013. SPARQL 1.1 Query Language. W3C Recommendation 21 March (2013). https://www.w3.org/TR/sparql11-query/
[5] Rakebul Hasan and Fabien Gandon. 2014. A Machine Learning Approach to SPARQL Query Performance Prediction. In *WI-IAT*. 266–273.
[6] Eugenie Yujing Lai, Zainab Zolaktaf, Mostafa Milani, Omar AlOmeir, Jianhao Cao, and Rachel Pottinger. 2023. Workload-Aware Query Recommendation Using Deep Learning. In *EDBT'23*. 53–65.
[7] Abiram Mohanaraj, Matteo Lissandrini, and Katja Hose. 2025. PlanRGCN: Predicting SPARQL Query Performance. *PVLDB* 18, 6 (2025), 1621–1634.
[8] Tomer Sagi, Matteo Lissandrini, Torben Bach Pedersen, and Katja Hose. 2022. A design space for RDF data representations. *VLDB J.* 31, 2 (2022), 347–373.
[9] Dimitris Tsesmelis and Alkis Simitsis. 2022. Database Optimizers in the Era of Learning. In *ICDE*. IEEE, 3213–3216.
[10] Hamada M. Zahera, Manzoor Ali, Mohamed Ahmed Sherif, Diego Moussallem, and Axel-Cyrille Ngonga Ngomo. 2024. Generating SPARQL from Natural Language Using Chain-of-Thoughts Prompting. In *SEMANTICS*, Vol. 60. 353–368.