Sapphire: Querying RDF Data Made Simple

Ahmed El-Roby[1] Khaled Ammar[1] Ashraf Aboulnaga[2] Jimmy Lin[1]

[1]University of Waterloo
[2]Qatar Computing Research Institute, HBKU

{aelroby, kammar, jimmylin}@uwaterloo.ca aaboulnaga@qf.org.qa

ABSTRACT

There is currently a large amount of publicly accessible structured data available as RDF data sets. For example, the Linked Open Data (LOD) cloud now consists of thousands of RDF data sets with over 30 billion triples, and the number and size of the data sets is continuously growing. Many of the data sets in the LOD cloud provide public SPARQL endpoints to allow issuing queries over them. These endpoints enable users to retrieve data using precise and highly expressive SPARQL queries. However, in order to do so, the user must have sufficient knowledge about the data sets that she wishes to query, that is, the structure of data, the vocabulary used within the data set, the exact values of literals, their data types, etc. Thus, while SPARQL is powerful, it is not easy to use. An alternative to SPARQL that does not require as much prior knowledge of the data is some form of keyword search over the structured data. Keyword search queries are easy to use, but inherently ambiguous in describing structured queries.

This demonstration introduces Sapphire, a system for querying RDF data that strikes a middle ground between ambiguous keyword search and difficult-to-use SPARQL. Our system does not replace either, but utilizes both where they are most effective. Sapphire helps the user construct expressive SPARQL queries that represent her information needs without requiring detailed knowledge about the queried data sets. These queries are then executed over public SPARQL endpoints from the LOD cloud. Sapphire guides the user in the query writing process by showing suggestions of query terms based on the queried data, and by recommending changes to the query based on a predictive user model.

1. INTRODUCTION

A large amount of publicly accessible structured data is available on the web in the RDF data sets that make up the Linked Open Data (LOD) cloud[1]. This is valuable data that can be useful in many different application domains. To enable users to retrieve this data, many of the data sets in the LOD cloud have SPARQL [8] endpoints that allow users to issue structured queries. In order to use these endpoints, a user must know the structure and vocabulary of the data set being queried, and the exact values of literals and their data types. At the scale of the LOD cloud, it is not convenient, and sometimes not possible, to have the level of knowledge of the schema elements and the data that is required to compose a SPARQL query over a data set. Thus, while SPARQL is powerful, it is not easy to use. An alternative to SPARQL that does not require as much prior knowledge of the data is some form of keyword search over the structured data. There has been a significant amount of research on querying information using keywords in both relational databases [1, 4] and RDF [5, 10]. Keyword search is very simple to use, but there is inherent ambiguity in using an unstructured keyword query to convey complex information needs. Keyword search cannot precisely specify structural properties required by the user, and cannot express commonly used functions provided by SPARQL such as ordering and aggregation.

(a)

```
SELECT ?name ?movies WHERE {
?movie name "Godfather".
?movie directedBy ?person.
?person name ?name.
?movies directedBy ?person.}
```

(b)

```
PREFIX dbp:  <http://dbpedia.org/property/>
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
SELECT ?directorName ?movies WHERE {
?movie dbp:name "The Godfather"@en.
?movie dbp:director ?director.
?director foaf:name ?directorName.
?movies dbp:director ?director.}
```

Figure 1: (a) A query written by the user to find the name of the director of the movie "Godfather" along with other movies he/she directed. (b) The correct SPARQL query that retrieves the required information from DBpedia.

In this demonstration, we introduce Sapphire, a system that strikes a balance between keyword search and SPARQL. Specifically, Sapphire helps the user construct SPARQL queries that represent her information needs, and uses techniques similar to keyword search to help her specify the elements of her query. To illustrate with an example, consider the queries in Figure 1. Figure 1(a) shows a SPARQL-like

[1] http://lod-cloud.net/

Proceedings of the VLDB Endowment, Vol. 9, No. 13
Copyright 2016 VLDB Endowment 2150-8097/16/09.

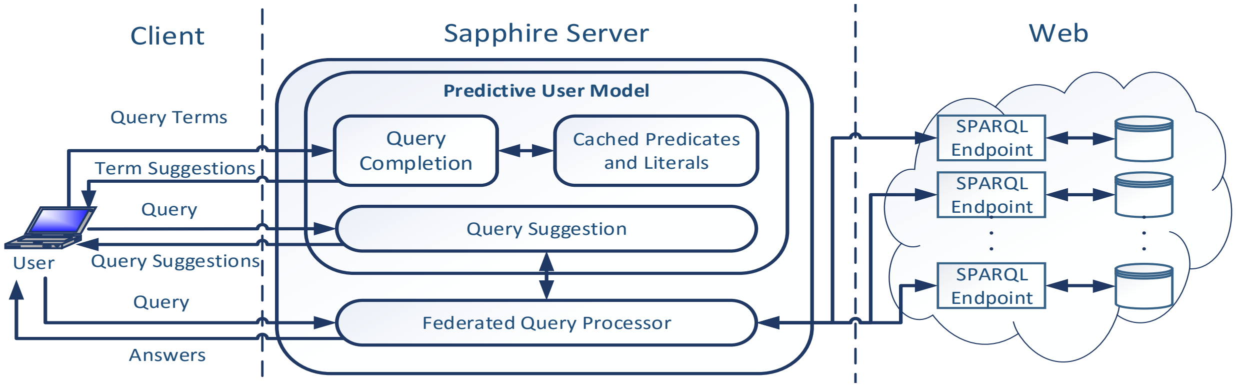


Figure 2: Architecture of Sapphire.

query that aims to find the director of the movie "Godfather" along with other movies that he/she directed. Figure 1(b) shows the correct SPARQL query that needs to be issued against DBpedia[2] to find this information. The goal of Sapphire is to start with a query as in Figure 1(a) and help the user construct the query in Figure 1(b). For example, the user knows that she needs to find the names of persons who are directors, but does not know the vocabulary used for these concepts (foaf:name and dbp:director). The user also knows that she is looking for the movie "Godfather", but does not know that the correct literal is "The Godfather"@en. Sapphire would provide this missing information. Sapphire would also execute the query over public SPARQL endpoints of LOD cloud data sets using a federated query processing engine so that answers from multiple data sets can be retrieved.

There has been prior work on helping users construct SPARQL queries. However, the focus of that work has been on users who cannot (or do not want to) directly express their information needs in SPARQL. For example, some work [3, 9] requires the user to issue keyword queries and uses these queries to create candidate SPARQL subqueries that are shown to the user to choose from and incrementally build the query she has in mind. The user is restricted to the subqueries that are automatically generated by the system, and needs multiple interactions to construct her query. Other work [6] requires the user to specify examples of data that should be in the query answer and creates a SPARQL query based on these examples. This approach requires the user to know enough about the data to specify example answers, and can only create SPARQL queries of limited complexity. In contrast, Sapphire is aimed at users who can express their information needs as SPARQL queries, but need help with specifying the details of these queries based on the data being queried.

Sapphire uses a *predictive user model* (PUM) that makes data-driven suggestions to the user as she writes the query. These suggestions are based on the keywords the user writes in the query. This can be described as issuing a keyword search (albeit with incomplete query terms) to find entities or relationships that the user is interested in. These suggestions are based on the similarity between query terms (after being enriched with synonyms and taxonomical relationships) and the actual data that the user is querying. When the user writes a valid query, Sapphire not only retrieves the answers to this query, but also uses its PUM to find and suggest changes to the issued query that may be relevant. By allowing the user to express structured queries while not necessarily being aware of the details of the structure and vocabulary of the queried data set, Sapphire bridges the gap between ambiguous keyword search approaches describing structured queries and complex query languages like SPARQL. Next, we present an overview of Sapphire and then discuss the demonstration scenario in Section 3.

2. OVERVIEW OF SAPPHIRE

2.1 System Architecture

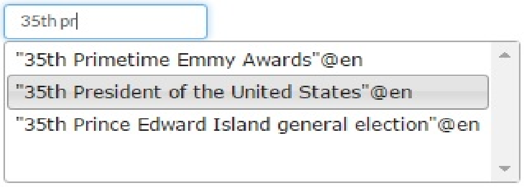
Figure 2 shows the system architecture of Sapphire. To initialize the system, the user inputs a set of SPARQL endpoints that she wishes to query. Sapphire goes through an initialization step in which it caches a subset of the predicates and literals from these endpoints on the server (Section 2.2). The Predictive User Model (PUM) module (Section 2.3) consists of the Query Completion Module (QCM) and the Query Suggestion Module (QSM). The QCM provides auto-complete suggestions while the user is typing a query. The suggestions are fetched from the cache of predicates and literals. The QSM suggests alternative queries that can replace the query input by the user. Another module in Sapphire is the Federated Query Processor, which executes queries over the available SPARQL endpoints and returns the answers (Section 2.4). The returned answers can be manipulated for easier consumption (Section 2.5).

2.2 Initialization

Prior to accepting queries, Sapphire goes through an initialization step in which it loads a subset of the predicates and literals from the data sets that will be queried. This subset is loaded locally into server memory so that it can be used by the QCM to provide suggestions to the user to complete the query. The subset is selected based on statistics over the length of the literals to avoid literals that are too short or too long and therefore unlikely to be part of a user query. The set of literals is partitioned by length into bins for faster search. The selected predicates and literals are loaded into memory at the Sapphire server for near-instant response when finding matches for input that the user provides for different parts of the query (e.g., in query form text boxes). This will be discussed in Section 2.3.

The initialization step is fast and the size of the cached predicates and literals is small compared to the size of the queried data set. For example, on approximately 18 GB

[2] http://dbpedia.org/sparql



35th pr

"35th Primetime Emmy Awards"@en

"35th President of the United States"@en

"35th Prince Edward Island general election"@en

Figure 3: Auto-complete suggestions using the QCM.

of DBpedia data, the initialization step takes less than 9 minutes, and the size of the cached predicates and literals is less than 4% of the size of the data set. Thus, Sapphire has low setup time and maintainability effort.

2.3 Predictive User Model

Query Completion Module. Sapphire helps users to construct SPARQL queries by providing almost-instant auto-complete suggestions and query answers. In Sapphire, the user is not required to write anything in the SELECT clause of the SPARQL query. The user inputs the triples that represent her information needs in the WHERE clause, and when the system detects one or more valid statements, it executes the query written so far and returns answers for all the variables in the query. The user can then continue writing other statements in the query or manipulate the answer table as discussed in Section 2.5. Providing a SELECT clause or adding modifiers to the query is optional, for example if the user needs to use aggregates like COUNT in the SELECT clause or query modifiers like GROUP BY.

In Sapphire, the user inputs the query via a web form, and as the user starts typing in a text box of this form, suggestions are shown to her instantly based on what she has typed so far. For example, if the user types "`?person ?job 35th pr`", suggestions that complete this pattern are shown almost instantly (see Figure 3). These suggestions are retrieved as follows: As the user types in the query, the term written so far is sent to the QCM, which searches in the cached set of predicates and literals for approximate matches. The set of cached literals is partitioned by length, so the search can eliminate partitions with literals that are too short or too long which improves performance significantly. The matcher uses Jaro-Winkler distance [2]. Any match with a score above a threshold θ is considered to be a candidate and the top k are shown to the user.

As the user types the query, answers that match the basic graph pattern that is written so far are displayed immediately. For example, as soon as the user chooses "`35th President of the United States`"`@en`, a table showing the values for the variables `?person` and `?job` are displayed.

For later triples in the query, choices made in previous triples are considered when returning suggestions to the user. For example, if the user chooses the highlighted suggestion in Figure 3, auto-complete suggestions for the next query triple that do not conform to this choice are given lower rank, and suggestions that conform to it are ranked higher. It is possible that the user may make incorrect choices and thus get incorrect answers. The Query Suggestion Module (QSM) that is described in the next section addresses this challenge.

Query Suggestion Module. After the user inputs a query, the query is validated and executed. Whenever a query is executed, the QSM tries to find alternatives to the query that was constructed by the user. Figure 4 shows an example of how the QSM suggests changes to the executed query. In this example, the user wants to find all people with the surname "Kennedys" (in plural form), which was one of the suggestions displayed by the QCM. However, no answers were found using this surname. The QSM suggests a modification that will result in finding 1,051 answers, by changing "Kennedys" to "Kennedy". If the user accepts this suggestion and updates the query, the new query is executed and the answers are displayed in the answer table (Figure 5). New suggestions are now displayed to the user in case these answers still do not satisfy her information needs.

The query alternatives are shown to the user in the form of suggestions to change one term at a time. For example, one suggestion could be "In the triple `(subject, predicate, object)`, did you mean `predicate'`, instead of `predicate`? There are N answers available.". This approach avoids showing the user a completely rewritten SPARQL query in one step, which would make the suggestions difficult to understand, especially for large and complex queries. The suggested queries are also executed in the background using the Federated Query Processor and answers are prefetched so that when the user decides to choose one of the alternatives, the query is not re-executed, and the answers are displayed almost-instantaneously.

The QSM finds a set of possible alternatives for each URI or literal in the query. We enrich the terms in the query with synonyms (obtained from WordNet[3]) to be able to find more alternatives. During the search, we also consider taxonomical differences (e.g., hypernyms and hyponyms) in order to discover structural differences between the query and the schema of the queried data set. Each match found for each URI or literal creates a new query. This results in a set of candidate queries that can be suggested as alternatives to the current query. This set is ranked based on: (1) The similarity score between the term in the query and the alternative term. The higher the similarity, the higher the rank of the candidate query. (2) The number of answers in the alternative query. The more the answers, the higher the rank of the candidate query. A weighted average is used to combine these two ranking criteria.

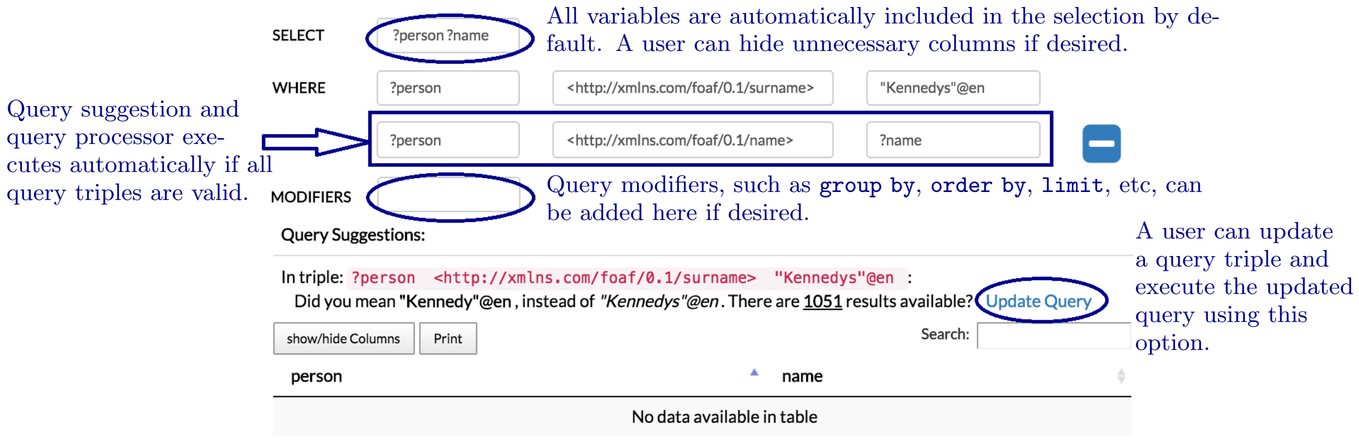
2.4 Federated Query Processor

The goal of Sapphire is to help users issue queries over a large number of RDF data sets without having complete knowledge about them. Therefore, when a valid SPARQL query is expressed, it is executed over all available data sets. This is done through the Federated Query Processor module, which splits the query and executes subqueries over relevant data sets through their respective SPARQL endpoints. It combines the answers from different sources and shows them to the user. Sapphire uses FedX [7] as the federated query processor. However, Sapphire does not use any features specific to FedX and can use any federated query processing system.

2.5 Manipulating Answers

When the answers to a query are displayed to the user, she has the ability to manipulate them in the answer table, as shown in Figure 5. Supported operations include the following: the user can search the answer table using a keyword search box, order the answers by any column, show and hide

[3] `https://wordnet.princeton.edu/`



SELECT	?person ?name		
WHERE	?person	<http://xmlns.com/foaf/0.1/surname>	"Kennedys"@en
	?person	<http://xmlns.com/foaf/0.1/name>	?name
MODIFIERS			

Query Suggestions:

In triple: `?person <http://xmlns.com/foaf/0.1/surname> "Kennedys"@en` :
Did you mean "Kennedy"@en , instead of *"Kennedys"@en*. There are 1051 results available? Update Query

person	name
No data available in table	

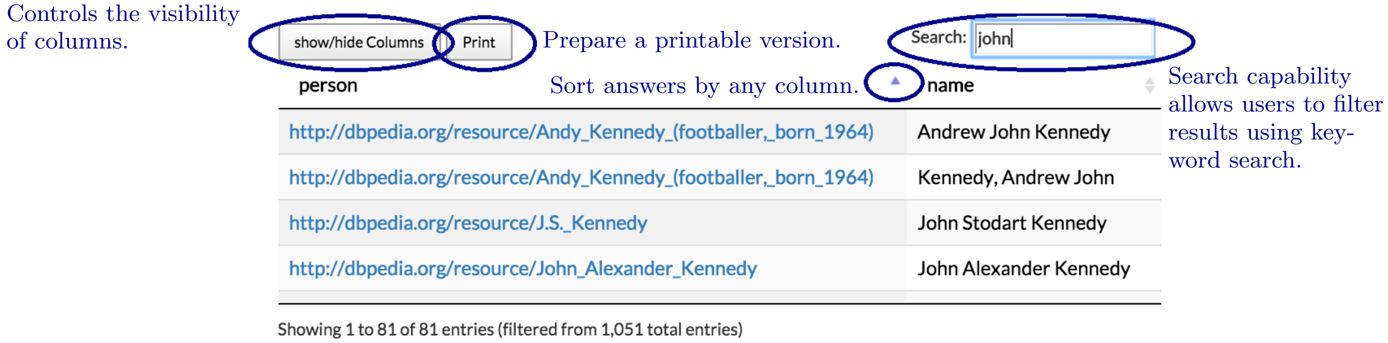
All variables are automatically included in the selection by default. A user can hide unnecessary columns if desired.

Query suggestion and query processor executes automatically if all query triples are valid.

Query modifiers, such as `group by`, `order by`, `limit`, etc, can be added here if desired.

A user can update a query triple and execute the updated query using this option.

Figure 4: Showing a suggestion to modify the current query that returned no answers.



person	name
http://dbpedia.org/resource/Andy_Kennedy_(footballer,_born_1964)	Andrew John Kennedy
http://dbpedia.org/resource/Andy_Kennedy_(footballer,_born_1964)	Kennedy, Andrew John
http://dbpedia.org/resource/J.S._Kennedy	John Stodart Kennedy
http://dbpedia.org/resource/John_Alexander_Kennedy	John Alexander Kennedy

Showing 1 to 81 of 81 entries (filtered from 1,051 total entries)

Controls the visibility of columns.

Prepare a printable version.

Sort answers by any column.

Search capability allows users to filter results using keyword search.

Figure 5: The answer table after applying the query suggestion in Figure 4. In this example, the 1,051 answers to the query are filtered via a keyword search on "`john`", and the filtered answers are ordered by the "`person`" column.

columns, and drag and drop answers from the answer table to the query text boxes for additional queries.

3. DEMONSTRATION SCENARIO

For the purpose of the demonstration, we will have Sapphire working with a set of SPARQL endpoints of different data sets from the LOD cloud (e.g., DBpedia, Geonames, LinkedMDB, etc.). The demonstration participants will be able to compose SPARQL queries that represent questions they have in mind. For example, *"Capitals of all countries in Africa"*, *"Construction date of the Statue of Liberty"*, *"Number of inhabitants of the largest city in Canada"*, etc. In addition, we will also have a set of such questions that participants can choose from. These questions will be based on the Question Answering over Linked Data benchmarks[4]. With no prior knowledge of the queried data sets, and even without knowing what data sets should be queried, the participants should be able to write valid SPARQL queries and get answers for their questions, manipulate the returned answers, explore the data, and ask new questions.

4. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: Enabling keyword search over relational databases. *SIGMOD*, 2002.

[2] W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string metrics for matching names and records. *IJCAI*, 2003.

[3] E. Demidova, X. Zhou, and W. Nejdl. A probabilistic scheme for keyword-based incremental query construction. *TKDE*, 24(3), 2012.

[4] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. *VLDB*, 2002.

[5] E. Kaufmann and A. Bernstein. How useful are natural language interfaces to the semantic web for casual end-users? *ISWC*, 2007.

[6] J. Lehmann and L. Bühmann. AutoSPARQL: Let users query your knowledge base. *ESWC*, 2011.

[7] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization techniques for federated query processing on linked data. *ISWC*, 2011.

[8] SPARQL 1.1 query language. http://www.w3.org/tr/sparql11-query/.

[9] G. Zenz, X. Zhou, E. Minack, W. Siberski, and W. Nejdl. From keywords to semantic queries-Incremental query construction on the semantic web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3), 2009.

[10] Q. Zhou, C. Wang, M. Xiong, H. Wang, and Y. Yu. Spark: Adapting keyword query to semantic search. *ISWC*, 2007.

[4] `http://qald.sebastianwalter.org/`