Detecting SPARQL Query Templates for Data Prefetching

Johannes Lorey and Felix Naumann

Hasso Plattner Institute,
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
{johannes.lorey,felix.naumann}@hpi.uni-potsdam.de

Abstract. Publicly available Linked Data repositories provide a multitude of information. By utilizing SPARQL, Web sites and services can consume this data and present it in a user-friendly form, e.g., in mash-ups. To gather RDF triples for this task, machine agents typically issue similarly structured queries with recurring patterns against the SPARQL endpoint. These queries usually differ only in a small number of individual triple pattern parts, such as resource labels or literals in objects. We present an approach to detect such recurring patterns in queries and introduce the notion of query templates, which represent clusters of similar queries exhibiting these recurrences. We describe a matching algorithm to extract query templates and illustrate the benefits of prefetching data by utilizing these templates. Finally, we comment on the applicability of our approach using results from real-world SPARQL query logs.

1 Introduction

Public SPARQL endpoints provide valuable resources for various information needs, e.g., drug information[1] or government spending data[2]. While end users are in most cases free to query these endpoints using Web forms, a much more widespread way to consume the provided data is through an intermediary software or service [14], including mash-ups[3,4] or general-purpose exploration[5] tools.

Whereas such frontends may increase usability, they typically reduce the scope of issued queries. Depending on the architecture and purpose of the software, requests exhibit certain recurring patterns [14], e.g., based on interaction with a fixed user interface. Potentially, these patterns result from combining Linked Data with unstructured or semi-structured information. For example, literals, such as labels or latitude and longitude specifications, may be extracted from user input and serve as objects of an individual triple pattern within a query, whereas the overall structure of this query is hard-coded. Hence, the application or Web site issues many highly-similar queries on behalf of its users

[1] http://www4.wiwiss.fu-berlin.de/drugbank/sparql
[2] http://govwild.hpi-web.de/sparql
[3] http://km.aifb.kit.edu/sites/spark/
[4] http://code.google.com/p/sgvizler/
[5] http://iwb.fluidops.com/

and utilizes only a subset of the information provided by the SPARQL endpoint. However, those (nearly) identical requests generated by user input increase the load on the SPARQL endpoint as well as the response time of the application's frontend. Additionally, in case the SPARQL endpoint becomes unavailable, the entire application has no access to the data.

One solution to this problem is to employ result caching. Caching eliminates the need to issue identical requests to the SPARQL endpoint multiple times assuming the knowledge base does not change over time. However, this solution works only if the exact same query is discovered in subsequent requests. In a real-world scenario however, it is more likely to encounter similar queries retrieving information about related resources. For these new queries, none or only partial locally cached information of previous requests can be used.

However, it might prove beneficial to gather the data relevant for related resources if a recurring access pattern is discovered. There exist different approaches of how to detect such related resources, including considering ontology information or graph distance metrics. In this work, we do not assume knowledge of such metadata and instead focus on structural elements of the SPARQL queries to determine the relatedness of RDF resources. We present an approach to detect recurring query patterns and use these patterns to introduce the notion of query templates. Query templates can be considered representatives of potentially overlapping query clusters containing structurally similar SPARQL queries. Furthermore, we introduce a use case for these query templates where the idea is to reduce the number of queries issued against a SPARQL endpoint by prefetching data relevant for subsequent requests.

This paper is organized as follows: We present related research in the fields of SPARQL query profiling and semantic caching in Sec. 2. In Sec. 3, we introduce fundamental notions required for this work. Section 4 provides details of our approach for discovering triple pattern mappings and graph pattern mappings as well as an algorithm for detecting and extracting query templates. We present some results for determining query templates in query sessions and evaluate our query rewriting approach on real-world SPARQL query logs in Sec. 5. Lastly, we conclude this work and comment on future research activities in Sec. 6.

2 Related Work

The related work for this paper draws mainly from two fields (i) *SPARQL Query Profiling*, e.g., the systematic analysis of queries to detect usage patterns, and (ii) *Semantic Caching and Prefetching*, e.g., techniques to either retain previously fetched data or retrieve data relevant for subsequent queries.

2.1 SPARQL Query Profiling

There have been a number of scientific projects aiming for a better understanding of structures and patterns of Linked Data. Here, most of the work has focused on profiling the data itself, such as [1, 4, 8]. However, analyzing and profiling actual

queries on Linked Data has recently also spawned a number of applications, such as SPARQL benchmarking [3, 12] or providing query suggestions [9, 16].

Our work is closely related to the latter topic. As the results in [14] suggest, there is great potential for discovering and reusing patterns of SPARQL queries. Indeed, in [9] the authors present a supervised machine learning framework to suggest SPARQL queries based on examples previously selected by the user. The authors claim that their approach benefits users who have no knowledge of the underlying schema or the SPARQL query language. A similar approach in [16] allows users to refine an initial query based on keywords.

In contrast to these works, the goal of our research is an automated approach to prefetch information without a priori knowledge of the knowledge base. Moreover, we rely on the structure of queries instead of applying natural language processing techniques on potentially unrelated keywords or resources. Additionally, we allow analysis of complex SPARQL queries and offer a means to cluster such queries for subsequent analysis. Overall, our research extends previous works on profiling Linked Open Data usage [11, 14] by suggesting a concrete use case for recurring patterns in SPARQL queries.

2.2 Semantic Caching and Prefetching

Semantic caching builds on the idea of maintaining a local copy of retrieved data that can be used for subsequent queries. As with traditional caching, one of the motivations for semantic caching is to reduce the transmission overhead when retrieving data over a network link. Conventional approaches, such as tuple or page caching, usually retain fetched data based on temporal or frequency aspects, e.g., by prioritizing least-recently or least-frequently used items. Such techniques also exist for SPARQL query result caching [10, 15]. Compared to this, semantic caching employs more fine-grained information to characterize data, e.g., in order to establish variable-sized semantic regions containing related tuples [5].

Closely related to semantic caching and our work is prefetching. Instead of simply retaining tuples retrieved previously, prefetching allows to gather data that is potentially useful for subsequent queries based on semantic information derived from past queries or the overall system state. In computer architecture design, prefetching is usually employed to request instructions that are anticipated to be executed in the future and place them in the CPU cache. For information retrieval, query prefetching typically assumes a probabilistic model, e.g., considering temporal features [6]. However, to the best of our knowledge, there have been no attempts to prefetch RDF data based on the structure of sequential related SPARQL queries within and across query sessions.

3 SPARQL Preliminaries

SPARQL is the de facto standard query language for RDF triples. In this section, we introduce some basic notions of SPARQL. Based on this, we illustrate several concepts used in this work to identify individual elements of a query. We use

these concepts in Sec. 4 to describe a matching algorithm for SPARQL queries based on an underlying query normal form.

3.1 SPARQL Graph Patterns

One central concept of a SPARQL query is that of a triple pattern $T = (s, p, o) \in (V \cup U) \times (V \cup U) \times (V \cup U \cup L)$ with V being a set of variables, U being a set of URIs, and L being a set of literals. A SPARQL query Q contains a number of graph patterns $P_1, P_2, \ldots$, which are defined recursively: (i) A valid triple pattern T is a graph pattern. (ii) If P_1 and P_2 are graph patterns, then P_1 `AND` P_2, P_1 `UNION` P_2, and P_1 `OPTIONAL` P_2 are graph patterns [13]. While there is the notion of empty graph patterns in SPARQL, we consider only non-empty graph patterns. Additionally, we focus on SELECT queries. An example of such a query is illustrated in Listing 1.

```
SELECT * WHERE {
  {
    ?p1 foaf:firstName "Alice" .
    ?p1 ?associationWith example:Bob .
  } UNION {
    ?p2 foaf:firstName "Carol" .
    OPTIONAL {
      ?p2 ?associationWith ?p1 .
    }
  }
}
```

Listing 1: SPARQL query example

In terms of relational operations, the keyword `AND` represents an inner join of the two graph patterns, `UNION` unsurprisingly denotes their union, and `OPTIONAL` indicates a left outer join between P_1 and P_2. Whereas `UNION` and `OPTIONAL` are reserved keywords in actual SPARQL queries to indicate the corresponding connection between two graph patterns, the `AND` keyword is omitted. In [13], it is shown that there exists a notion of a normal form for SPARQL queries based on the recursive graph pattern structure presented earlier and the precedence of the operators connecting those graph patterns. Hence, for this work we assume a SPARQL SELECT query can always be expressed as a composition of graph patterns, connected either by `UNION`, `AND`, or `OPTIONAL`.

Curly braces delimiting a graph pattern (i.e., $\{P\}$) are syntactically required for both P_1 and P_2 in a `UNION` statement and only for P_2 in an `OPTIONAL` statement. Furthermore, we refer to the largest delimited graph pattern P contained in a SPARQL query Q as the *query pattern* P_Q. Note that every query has exactly one query pattern P_Q. To increase readability and avoid confusion with set braces, we omit the brace delimiters in this work whenever possible. For the remainder of this work, P_i denotes a valid graph pattern contained in P_Q.

In Sec. 4, we introduce a matching algorithm for graph patterns. One necessary prerequisite for this algorithm is to identify individual child graph patterns

contained in P_Q. For example, the query in Listing 1 contains the following three non-trivial child graph patterns $P_{\texttt{AND}}$, $P_{\texttt{OPTIONAL}}$, and $P_{\texttt{UNION}}$:

$P_{\texttt{AND}} :=$
```
?p1 foaf:firstName "Alice" .
?p1 ?associationWith example:Bob .
```

$P_{\texttt{OPTIONAL}} :=$
```
?p2 foaf:firstName "Carol" .
OPTIONAL {
  ?p2 ?associationWith ?p1 .
}
```

$P_{\texttt{UNION}} = P_Q := P_{\texttt{AND}}$ `UNION` $P_{\texttt{OPTIONAL}}$

3.2 Graph Pattern Decomposition

To extract child graph patterns, we introduce the three functions $\Theta_{\texttt{UNION}}(P)$, $\Theta_{\texttt{AND}}(P)$, and $\Theta_{\texttt{OPTIONAL}}(P)$. They each take as input a graph pattern P and totally decompose P into the set of its non-empty child graph patterns $P_1, P_2, \ldots, P_n$, all conjoined exclusively by `UNION`, `AND`, or `OPTIONAL`, respectively. The three functions can then be applied recursively on the individual elements $P_1, P_2, \ldots, P_n$ in the result set, possibly yielding further non-trivial results.

For example, if we apply $\Theta_{\texttt{UNION}}(P_Q)$ on the query pattern in Listing 1, we retrieve the set $\{P_{\texttt{AND}}, P_{\texttt{OPTIONAL}}\}$. Similarly, $\Theta_{\texttt{AND}}(P_{\texttt{AND}})$ retrieves a set containing the two triple patterns listed above as elements. If no such total decomposition can be derived, the result set is empty, e.g., $\Theta_{\texttt{AND}}(P_Q) = \emptyset$ or $\Theta_{\texttt{UNION}}(P_{\texttt{AND}}) = \emptyset$.

Whereas in general, for $\oplus \in \{\texttt{UNION}, \texttt{AND}, \texttt{OPTIONAL}\}$:

$$\Theta_{\oplus}(P) \neq \emptyset \Leftrightarrow P := P_1 \oplus P_2 \oplus \ldots \oplus P_n,$$

the individual functions are defined as follows (all $n \geq 2$):

$$\Theta_{\texttt{UNION}}(P) := \begin{cases} \{P_1, \ldots, P_n\}, & \text{iff } P := P_1 \texttt{ UNION } P_2 \texttt{ ... UNION } P_n \\ \emptyset, & \text{else.} \end{cases}$$

$$\Theta_{\texttt{AND}}(P) := \begin{cases} \{P\}, & \text{iff } P \text{ is a triple pattern} \\ \{P_1, \ldots, P_n\}, & \text{iff } P := P_1 \texttt{ AND } P_2 \texttt{ ... AND } P_n \\ \emptyset, & \text{else.} \end{cases}$$

$$\Theta_{\texttt{OPTIONAL}}(P) := \begin{cases} \{P_1, \ldots, P_n\}, & \text{iff } P := P_1 \texttt{ OPTIONAL } P_2 \texttt{ ... OPTIONAL } P_n \\ \emptyset, & \text{else.} \end{cases}$$

We also define the function $\Theta(P)$ as a convenience method to detect whether for a graph pattern P a decomposition exists for either $\Theta_{\texttt{UNION}}(P)$, $\Theta_{\texttt{OPTIONAL}}(P)$, or $\Theta_{\texttt{AND}}(P)$ (in this order):

$$\Theta(P) := \begin{cases} \Theta_{\texttt{UNION}}(P), & \text{iff } \Theta_{\texttt{UNION}}(P) \neq \emptyset \\ \Theta_{\texttt{OPTIONAL}}(P), & \text{iff } \Theta_{\texttt{OPTIONAL}}(P) \neq \emptyset \\ \Theta_{\texttt{AND}}(P), & \text{else.} \end{cases}$$

Except for when P is a triple pattern and we apply $\Theta_{\texttt{AND}}(P) = P$, we also assume that all decompositions are non-trivial, i.e., $\Theta_{\oplus}(P) \neq \{P\}$. Hence, according to the underlying graph pattern normal form, all the above cases are mutually exclusive. We call $|P| = |\Theta(P)|$ the *size* of a graph pattern.

In addition, we introduce the function $\kappa(P)$ for a graph pattern P:

$$\kappa(P) := \begin{cases} \texttt{UNION}, & \text{iff } \exists P_1 \in P_Q : P \in \Theta_{\texttt{UNION}}(P_1) \\ \texttt{OPTIONAL}, & \text{iff } \exists P_1, P_2 \in P_Q : P, P_2 \in \Theta_{\texttt{OPTIONAL}}(P_1) \wedge P_2 \texttt{ OPTIONAL } P \\ \texttt{AND}, & \text{else.} \end{cases}$$

The function $\kappa(P)$ allows to determine how P is connected to other graph patterns in a graph pattern decomposition, e.g., $\forall P_i \in \Theta_{\texttt{UNION}}(P) : \kappa(P_i) = \texttt{UNION}$. We incorporate the results from both $\kappa(P)$ and $\Theta(P)$ in the algorithm presented in the next section. This information allows us to decide whether two graph patterns can be matched to one another or not.

4 Query Templates

In real-world applications, a large number of queries processed by a SPARQL endpoint exhibit similar structures and vary only in a certain number of resources. In this section, we present query templates that can be used to cluster these similar SPARQL query structures. To identify such query structures, we present a triple pattern similarity measure that is used for our recursive graph pattern matching algorithm. If the algorithm detects a match between the query patterns of two queries, they share a common query template.

4.1 Triple Pattern Similarity and Merging

We first define similar triple patterns that can be mapped to and merged with one another. To establish a mapping between two triple patterns $T_1 = (s_1, p_1, o_1)$ and $T_2 = (s_2, p_2, o_2)$, we try to match the individual elements of T_1 with the corresponding part of T_2, i.e., we align x_1 with x_2 for $x \in \{s, p, o\}$. To calculate the distance of such mappings, we introduce the score $\Delta(x_1, x_2)$:

$$\Delta(x_1, x_2) := \begin{cases} \dfrac{d(x_1, x_2)}{max(|x_1|, |x_2|) + 1} * k, & \text{if } x_1 \in V \wedge x_2 \in V, \text{ with } 0 \leq k < 1 \\ \dfrac{d(x_1, x_2)}{max(|x_1|, |x_2|) + 1}, & \text{if } (x_1 \in U \wedge x_2 \in U) \vee (x_1 \in L \wedge x_2 \in L) \\ 1, & \text{else.} \end{cases}$$

Here, V, U, L are the sets of variables, URIs, and literals, respectively, $|x|$ is the string length of x and $d(x_1, x_2) \rightarrow \mathbb{N}_0$ is a string distance measure with $d(x_1, x_2) = 0 \Leftrightarrow x_1 = x_2$. In our work, we use the Levenshtein distance. Notice that we apply the Levenshtein distance on the entire resource strings, i.e., including possible prefix definitions for URIs or types for literals.

If two queries are identical in structure and content except for their variable names, the binding result set of those variables retrieved from a SPARQL endpoint is the same. Hence, we assume that variables can be mapped more easily to one another than URIs or literals, and apply a factor $k \leq 1$ to $\Delta(x_1, x_2)$, if both x_1 and x_2 are variables. In our implementation, we use $k = \frac{1}{3}$.

To evaluate how easily two triple patterns can be merged, we introduce the triple pattern distance score $\Delta(T_1, T_2)$ that sums up the individual distance scores, i.e., $\Delta(T_1, T_2) := \Delta(s_1, s_2) + \Delta(p_1, p_2) + \Delta(o_1, o_2)$.

```
1 ?p1 foaf:firstName "Alice" .
2 ?p1 ?associationWith example:Bob .
3 example:Bob foaf:firstName "Bob" .
4 example:Bob foaf:lastName "Alice" .
5 ?p2 foaf:firstName "Carol" .
6 ?p2 ?associationWith ?p1 .
```

Listing 2: Triple pattern similarity example

Consider the first triple pattern T_1 in Listing 2: The minimum distance score between T_1 and all other triple patterns shown, i.e., $min(\Delta(T_1, T_2), \ldots, \Delta(T_1, T_6))$, is $(\frac{1}{12} + 0 + \frac{5}{8}) \approx 0.71$ for T_5. For T_2, the minimum value is $(\frac{1}{12} + 0 + 1) \approx 1.08$ for T_6, and for T_3 it is $(0 + \frac{3}{15} + \frac{5}{8}) \approx 0.83$ for T_4.

We also allow the calculation of distance scores between two graph patterns P_1, P_2 as follows:

$$\Delta(P_1, P_2) := \begin{cases} \Delta(T_1, T_2), & \text{if } \Theta(P_1) = \{T_1\} \wedge \Theta(P_2) = \{T_2\} \\ \infty, & \text{else.} \end{cases}$$

This notation mainly serves as a shorthand for analyzing graph patterns with size 1, i.e., graph patterns that constitute triple patterns.

Finally, we introduce the generalization function $\lambda(T_1, T_2) = \hat{T}$ that takes as input two triple patterns T_1, T_2 and merges them into one $\hat{T} = (\hat{s}, \hat{p}, \hat{o})$. It does so by replacing the non-equal triple pattern elements between $T_1 = (s_1, p_1, o_1)$ and $T_2 = (s_2, p_2, o_2)$ with arbitrary, uniquely named variables. More formally, we first define $\lambda(x_1, x_2)$ on the triple pattern parts with $x \in \{s, p, o\}$:

$$\lambda(x_1, x_2) := \begin{cases} x_1, & \text{if } \Delta(x_1, x_2) = 0 \\ ?var, & \text{else.} \end{cases}$$

Here, $?var$ represents a variable unique to both triple patterns. The distance of any two of these introduced variables $\Delta(?var_1, ?var_2) = 0$. Thus,

$$\lambda(T_1, T_2) := (\lambda(s_1, s_2), \lambda(p_1, p_2), \lambda(o_1, o_2))$$

In particular, this means that $\hat{T} = T_1$ iff $\Delta(T_1, T_2) = 0$, i.e., no merging is necessary, if the two triple patterns are identical. As with Δ, we use the shorthand notation $\lambda(P_1, P_2)$, if $|P_1| = |P_2| = 1$.

4.2 Graph Pattern Matching

Using the triple pattern distance notion, we can now derive matchings between graph pattern by mapping their individual triple patterns. We consider the task of finding a match a variation of the stable marriage problem [7], which we solve greedily using Algorithm 1. The recursive algorithm takes as arguments two graph patterns P_1, P_2, a maximum distance threshold Δ_{max} for mapping any two triple patterns, and an existing mapping between triple patterns. This mapping is initially empty and is established in polynomial time by iterating over all graph patterns contained in P_1 and P_2. If no complete matching between P_1 and P_2 can be derived, the result of the algorithm is an empty set of mappings.

Two necessary conditions for a match are $\kappa(P_1) = \kappa(P_2)$ and $|\Theta(P_1)| = |\Theta(P_2)|$ (Line 2). Hence, the algorithm does not establish a match between graph patterns with different keywords or sizes. These conditions are necessary, as there might exist partial (i.e., non-perfect) matches between graph patterns of different sizes, but we are interested in discovering only complete matches.

The algorithm traverses over the graph patterns P_1^i contained in S_1 (which is initialized with the results of $\Theta(P_1)$) and tries to match these graph patterns with the graph patterns P_2^j in S_2 (comprising the results of $\Theta(P_2)$) (Line 7). In case both graph patterns currently in consideration have size 1, i.e., they are triple patterns (Line 8), the algorithm checks whether a mapping can be established between these two triple patterns.

Given that P_1^i and P_2^j exhibit the same keyword (Line 9), a mapping between the two triple pattern can be established under two conditions: (i) $\Delta(P_1^i, P_2^j) \leq \Delta_{max}$ and there is currently no other mapping between P_2^j and another triple pattern (Line 12), or (ii) the current mapping of P_2^j has a higher distance score to it than $\Delta(P_1^i, P_2^j)$ (Line 17). In the first case, the mapping is established, in the second case, the existing mapping is changed accordingly, and the previously mapped element P_1^* is again added to S_1 (Line 19). This ensures that the algorithm tries to discover a new match for P_1^* in a subsequent iteration. In both cases, the algorithm sets the value of the Boolean variable $foundMapping$ to `true` and continues by examining the next element in S_1.

If P_1^i and P_2^j are not triple patterns, i.e., their size is greater than 1, the algorithm is executed recursively, using P_1^i and P_2^j along with *mappings* as arguments (Line 24). If *mappings* has changed, either because there were new mappings added or previous mappings altered, $foundMapping$ is set to `true`.

If throughout this iteration, no mapping was discovered between P_1^i and P_2^j, i.e., $foundMapping$ is `false`, the returned mappings are empty (Line 28). Potentially, some mappings could have been determined throughout the recursion and added to *mappings*, while overall the current graph pattern P_1^i cannot be matched to any other graph pattern. To avoid partial matches, *mappings* is returned only if matches were established for all child graph patterns.

As mentioned earlier, Algorithm 1 determines mappings between two triple patterns T_1, T_2 only if they reside in graph patterns P_1, P_2 with identical keyword and size (Line 2). While there might exist a triple pattern T_i in another graph

Algorithm 1: GraphPatternMatching

Input : P_1, P_2 : Two graph patterns
Input : Δ_{max} : Triple pattern distance threshold
Input : *mappings* : Current triple pattern mappings
Output: *mappings* : Symmetric triple pattern mappings between P_1, P_2

```
1  S_1 ← Θ(P_1), S_2 ← Θ(P_2)
2  if κ(P_1) ≠ κ(P_2) ∨ |S_1| ≠ |S_2| then
3      return ∅
4  while S_1 ≠ ∅ do
5      P_1^i ← S_1.pollFirst()
6      foundMapping ← false
7      foreach P_2^j ∈ S_2 do
8          if |P_1^i| = 1 ∧ |P_2^j| = 1 then
9              if κ(P_1^i) = κ(P_2^j) then
10                 P_1^* ← mappings.get(P_2^j)
11                 if P_1^* = NIL then
12                     if Δ(P_1^i, P_2^j) ≤ Δ_max then
13                         mappings.put(P_2^j, P_1^i)
14                         foundMapping ← true
15                         break
16                 else
17                     if Δ(P_1^i, P_2^j) < Δ(P_1^*, P_2^j) then
18                         mappings.put(P_2^j, P_1^i)
19                         S_1.add(P_1^*)
20                         foundMapping ← true
21                         break
22         else
23             oldMappings ← mappings
24             mappings ← GraphPatternMatching(P_1^i, P_2^j, mappings)
25             if mappings ≠ ∅ ∧ mappings ≠ oldMappings then
26                 foundMapping ← true
27     if ¬foundMapping then
28         return ∅
29 return mappings
```

pattern P_i with $i > 2$ and a lower distance score $\Delta(T_1, T_i) < \Delta(T_1, T_2)$, these cannot be mapped, e.g., because of different keywords $\kappa(P_1) \neq \kappa(P_i)$.

Hence, any non-empty set of mappings resulting from Algorithm 1 can be considered stable in the sense that the mapped triple patterns have minimal distance to their mapping partner with respect to the graph pattern they are contained in. If there exists another possible mapping with a lower distance score for a particular triple pattern, this mapping would have been established instead of the current one (Line 17). Note however that the algorithm prefers the first

possible triple pattern mapping over all other possible mappings with identical triple pattern distance (Lines 15 and 21). If for any evaluated graph pattern no match could be determined, the overall return value of the algorithm is an empty set of mappings (Line 28). Conversely, any non-empty mapping result is complete (or perfect) and therefore maximal (the size of non-empty mappings is determined by the number of triple patterns contained in the graph pattern).

4.3 Query Templates and Clusters

Using the output of Algorithm 1, we can now discover *query templates*. The idea of query templates builds on the findings discussed in [14], where the authors mine SPARQL query log files to determine the behavior of agents issuing the respective query. We extend this approach by establishing a formal definition of what constitutes a query template and how to find it. In contrast to previous work, we also show a concrete application of query templates in the next section.

To generate a query template, we evaluate the mappings generated by GraphPatternMatching$(P_{Q_1}, P_{Q_2}, 1, \emptyset)$ for two SPARQL queries Q_1, Q_2 with query patterns P_{Q_1}, P_{Q_2}, respectively. If the output of Algorithm 1 is empty, no query template can be derived. Otherwise, we initialize the query template $\hat{Q}$ with the query Q_1 and replace all triple patterns T_i in $\hat{Q}$ with the merged triple pattern $\hat{T}$ that resulted from $\lambda(T_i, T_j)$ where $(T_i, T_j) \in map$. Whereas in general we require any introduced variable to be unique with relation to other variables in both P_{Q_1}, P_{Q_2}, if we observe a repeated merge between two identical triple pattern parts, e.g., two consecutive $\lambda(s_i, s_j) \neq s_i$, we re-use the variable introduced for the first merge. Finally, we consider $\hat{Q}$ to be a query template, if $\hat{Q} \neq Q_1$, i.e., $Q_1 \neq Q_2$ and at least one triple pattern mapping $(T_1, T_2) \in map$ is non-trivial.

Recall that variables introduced during merging have distance 0 to each other as defined in Subsec. 4.1. Hence, if we determine two query templates that are identical except for the identifiers of the introduced variables, we consider them to represent the same template. Thus, all queries sharing a query template form a *query cluster*, which may overlap. Notice that all queries in a cluster can be matched to the cluster's query template, albeit potentially only for $\Delta_{max} > 1$. We assume that for most queries in such a cluster a single resource or literal is replaced throughout all triple patterns as indicated by the findings in [14].

5 Evaluation

To evaluate our template discovery approach we analyzed the DBpedia 3.6 query log files from the USEWOD 2012 dataset [2]. In total, these files contain around 8.5 million anonymized queries received by the public DBpedia endpoint[6] on 14 individual days in 2011. We chose these particular log files mainly for three reasons: (i) the query intention is to some extent comprehensible to non-domain experts, (ii) the log files exhibit recurring query patterns [14], and (iii) all queries

[6] `http://dbpedia.org/sparql`

are assigned a source (hashed IP address) and timestamp (hourly granularity), allowing us to to coarsely delimit query sessions.

To illustrate the last point, an excerpt of the query log file *2011-01-24.log* is presented in Listing 3. Each line starts with the hashed IP address of the issuing source, followed by the timestamp and the actual query. We found that additional metadata provided in the log files, e.g., the user agent sending the requests, did not provide any information relevant to our work.

Listing 3 also indicates that the level of granularity in the query log is hours. For our experiments, we consider all queries from one user within one hour (i.e., with the same timestamp) to constitute a query session. Moreover, we also map queries in such a query session to the clusters they belong to. Hence, for the rest of our evaluation, query sessions can be considered sequences of query templates uniquely identified by a timestamp and user id.

```
1 237fbf63e8449c1ade56eb7d208ce219 - [24/Jan/2011 01:00:00 +0100] "/sparql/?query..." 200 512 "-" "-"
2 f452f4195b4d2046c77ad98496c1b127 - [24/Jan/2011 01:00:00 +0100] "/sparql/?query..." 200 1024 "-" "Java"
3 9b1d83195dd251275c55c12ac2efa43d - [24/Jan/2011 02:00:00 +0100] "/sparql/?query..." 200 512 "-" "Mozilla"
4 f452f4195b4d2046c77ad98496c1b127 - [24/Jan/2011 02:00:00 +0100] "/sparql/?query..." 200 1024 "-" "-"
```

Listing 3: Abbreviated excerpt from query log file *2011-01-24.log*

5.1 Query Session Analysis

For our first evaluation, we analyze the size, frequency, and contents of query sessions across all users. Figure 1 illustrates how often query sessions of different length occur. We distinguish between homogeneous query sessions, i.e., sessions containing only queries from the same cluster, and heterogeneous query sessions, i.e., sessions containing queries from at least two clusters. Overall, homogeneous query sessions occur far more often than heterogeneous query sessions, even if query sessions of length 1 (which are always homogeneous) are disregarded. Generally, the average length of homogeneous query sessions is also in order of magnitudes higher than the average length of heterogeneous query sessions.

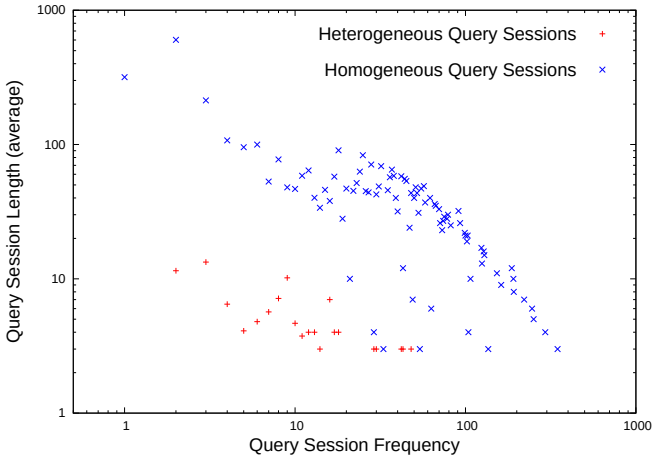
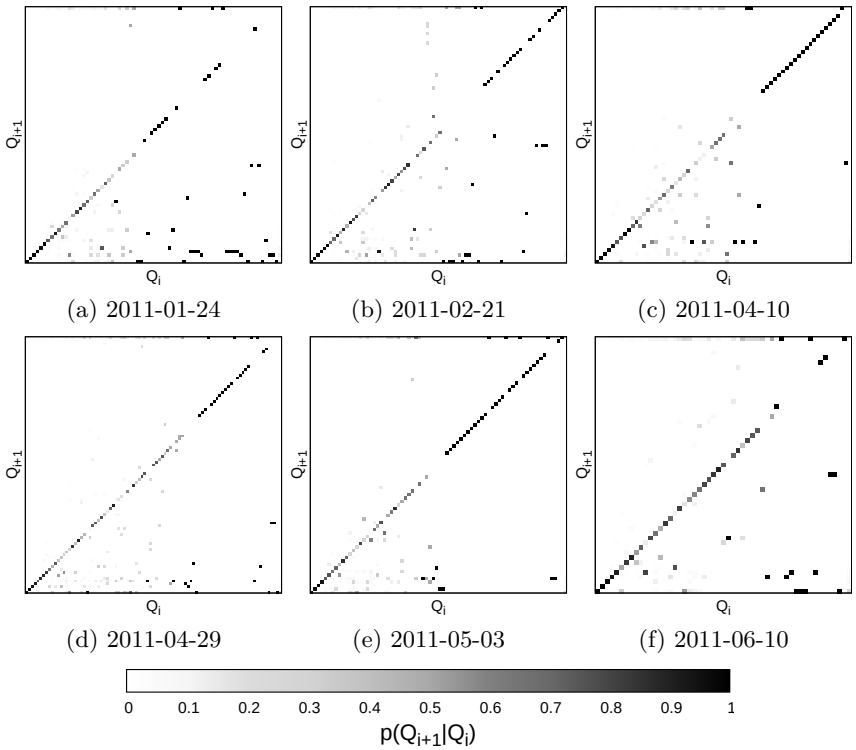


Fig. 1: Length of query sessions correlated with their frequency

Both these findings, i.e., the high frequency and length of homogeneous query sessions, indicate that most requests received by the DBpedia endpoint are similarly-structured SPARQL queries, most likely issued by machine agents. On the other hand, only a small percentage of relatively short query sessions are heterogeneous, possibly indicating human users querying the DBpedia endpoint.

We also evaluated the conditional probability of sequences of length 2 for all query clusters discovered in the log files and present results in Fig. 2. Here, both axes Q_i and Q_{i+1} of all individual diagrams correspond to the query clusters, where a single tick mark on each axis represents one cluster. Both axes are sorted in descending size of the represented query clusters. The values for $p(Q_{i+1}|Q_i)$ illustrate the probability of observing a query from a certain cluster given the cluster of the previous query. A high value (represented by a darker color) indicates that queries from two query clusters are likely to occur in sequence.

While the plots differ slightly for the various dates in Fig. 2, two general trends can be observed: First, the matrix of all conditional probabilities is sparsely populated, i.e., for a query belonging to any given query cluster discovered in the log, the subsequent query usually belongs to one of a limited number of clusters. In addition, there is a high probability that queries from one query cluster are followed by queries of the same cluster. This notion is illustrated by the high values on the diagonal of all diagrams.



(a) 2011-01-24 (b) 2011-02-21 (c) 2011-04-10

(d) 2011-04-29 (e) 2011-05-03 (f) 2011-06-10

Fig. 2: Conditional probabilities for sequences of length 2 of query templates

5.2 Query Template Prefetching

If many individual similarly-structured queries, i.e., queries from the same query cluster, are issued in immediate succession by an agent, as observed in Sec. 5.1, this agent essentially utilizes only parts of the provided data. Moreover, while the relevant information is retrieved one query at a time, it could instead be gathered all at once and used to populate a locally materialized view on the knowledge base. This approach yields advantages for both the endpoint provider and the data consumer by reducing the number of connections on the SPARQL endpoint and eliminating latency overhead (e.g., for query planning, disk access, and data transmission) on each request, respectively.

Consider the sample query in Listing 4: This query retrieves the English language abstract of the resource `dbpedia:Charreada`. Similar queries concerning abstracts were discovered around 3.5 million times for varying subjects in the query log files. The longest individual query sequence consisting only of distinct queries from this cluster issued by a single user contains 56,633 queries. At the time of writing, the DBpedia endpoint provides English abstract information for 3,769,926 resources. Hence, during the longest query sequence around 1.5% of all English DBpedia abstracts are retrieved. We discovered similar request patterns for other query sessions of this user and among query sequences of other users.

```
SELECT ?abstract WHERE {
  dbpedia:Charreada dbpedia-owl:abstract ?abstract .
  FILTER (langMatches(lang(?abstract), "en"))
}
```

Listing 4: SPARQL sample query retrieving the English abstract of a resource

To evaluate the accumulated latency overhead caused by such a large amount of similar queries, we first randomly extracted 100 sample queries from the query cluster containing requests retrieving English abstracts of a resource. Then, we sent these queries to the public SPARQL endpoint. Based on our measurements, the average time between issuing a query and receiving a result was around 5.2 ms. Retrieving the abstracts of 1000 resources using the query template on the other hand took only around 611 ms. Hence, issuing a single query template to retrieve results for related resources instead of multiple queries each retrieving only bindings for one resource leads to an execution speedup of nearly factor 10. For different query templates, we measured similar speedup results.

The benefit of prefetching data for future queries depends on how many queries actually exploit this locally available data. This number is influenced by the length of the analyzed time frame. We illustrate the advantages of prefetching for distinct query sessions in Tab. 1a and for all queries from a specific user within one day in Tab. 1b. Here, we chose the five users (identified by their abbreviated IP hash) with the most queries in the respective time frame. We identified the most common query cluster in this query set, gathered results for the corresponding template, and materialized these results locally. The coverage rate describes how many of these prefetched results were also retrieved individually by the queries within the respective time frame. Higher coverage indicates that more prefetched results were retrieved by actual queries.

Table 1: Template coverage rates for the top five users with the most queries

(a) Distinct query sessions

User ID	#Queries	Coverage Rate
237...	6,081	54.21%
ea0...	4,951	44.14%
6cb...	3,216	28.67%
e36...	3,106	27.69%
a40...	455	4.05%

(b) All queries within a day

User ID	#Queries	Coverage Rate
237...	68,472	100.00%
f45...	29,235	21.11%
6cb...	18,844	100.00%
5de...	13,500	100.00%
499...	9,747	27.84%

Table 1 illustrates that for a large number of queries issued over a short period of time by a distinct user, i.e., a single hour or day, a local cache containing the data retrieved in advance can efficiently provide results for these queries. This effect becomes more obvious for longer time periods: As Tab. 1b indicates, there are cases when prefetched data can be used for myriads of queries on a single day and all prefetched information is completely utilized during this time frame.

6 Conclusion

In this work, we presented the notion of SPARQL query templates. They represent potentially overlapping clusters of similarly-structured queries, where all elements within a cluster exhibit recurring query patterns and are subsumed by the template. We described an algorithm to detect and extract query templates based on a flexible resource similarity distance function. Furthermore, we evaluated our approach on real-world SPARQL query logs. Here, we discovered three main results: First, the large amount of SPARQL queries received by the DBpedia endpoint can be mapped to a small number of query clusters. In addition, resulting query sessions are mostly homogeneous, i.e., queries from a specific cluster are likely to be followed by queries from the same cluster. Lastly, retrieving combined results for queries from the same cluster instead of issuing individual queries reduces the latency overhead.

We have illustrated a specific use case for query templates by exploiting these findings: Result prefetching. Here, instead of issuing multiple queries from the same cluster, we instead issue the common query template that subsumes these queries. As we have shown in our evaluation, this is particularly useful for longer query sessions. If we assume that a cache containing these prefetched results is maintained in-between query sessions, even more cache hits are generated.

As the findings in this work have proven, there is a huge potential for retrieving semantically relevant data for future queries before these are actually issued. Whereas the introduced approaches already work well for long query sessions or across query sessions, they do not cater to short query sessions with mixed requests, typically encountered when human agents issue exploratory SPARQL queries. Thus, in future work we plan to extend our query prefetching approach by adapting more sophisticated query rewriting approaches based on common information retrieval strategies for RDF data. Ultimately, our goal is to train a classifier to automatically choose the most suitable of these rewriting methods.

References

1. Bartolomeo, G., Salsano, S.: A spectrometry of linked data. In: Proceedings of the WWW Workshop on Linked Data on the Web (LDOW), Lyon, France (2012)
2. Berendt, B., Hollink, L., Hollink, V., Luczak-Rösch, M., Möller, K., Vallet, D.: USEWOD2012 – 2nd international workshop on usage analysis and the web of data. In: Proceedings of the International World Wide Web Conference (WWW), Lyon, France (2012)
3. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. International Journal on Semantic Web and Information Systems **5**(2) (2009) 1–24
4. Böhm, C., Lorey, J., Naumann, F.: Creating voiD descriptions for web-scale data. Journal of Web Semantics **9**(3) (2011) 339–345
5. Dar, S., Franklin, M.J., Jónsson, B.T., Srivastava, D., Tan, M.: Semantic data caching and replacement. In: Proceedings of the International Conference on Very Large Databases (VLDB), Bombay, India (1996) 330–341
6. Fagni, T., Perego, R., Silvestri, F., Orlando, S.: Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. ACM Transactions on Information Systems **24**(1) (2006) 51–78
7. Gale, D., Shapley, L.S.: College admissions and the stability of marriage. The American Mathematical Monthly **69**(1) (1962) 9–15
8. Khatchadourian, S., Consens, M.P.: Exploring RDF usage and interlinking in the linked open data cloud using explod. In: Proceedings of the WWW Workshop on Linked Data on the Web (LDOW). (2010)
9. Lehmann, J., Bühmann, L.: AutoSPARQL: Let users query your knowledge base. In Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., Leenheer, P., Pan, J., eds.: The Semantic Web: Research and Applications. Volume 6643 of LNCS. Springer Heidelberg (2011) 63–79
10. Martin, M., Unbehauen, J., Auer, S.: Improving the performance of semantic web applications with sparql query caching. In Aroyo, L., Antoniou, G., Hyvönen, E., Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T., eds.: The Semantic Web: Research and Applications. Volume 6089 of LNCS. Springer Heidelberg (2010) 304–318
11. Möller, K., Hausenblas, M., Cyganiak, R., Grimnes, G.A.: Learning from linked open data usage: Patterns & metrics. In: Proceedings of the Web Science Conference, Raleigh, NC, USA (2010)
12. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.C.: DBpedia SPARQL benchmark – performance assessment with real queries on real data. In Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E., eds.: The Semantic Web – ISWC 2011. Volume 7031 of LNCS. Springer Heidelberg (2011) 454–469
13. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Transactions on Database Systems (TODS) **34**(3) (2009) 16:1–16:45
14. Raghuveer, A.: Characterizing machine agent behavior through SPARQL query mining. In: Proceedings of the International Workshop on Usage Analysis and the Web of Data, Lyon, France (2012)
15. Yang, M., Wu, G.: Caching intermediate result of SPARQL queries. In: Proceedings of the International World Wide Web Conference (WWW), Hyderabad, India (2011) 159–160
16. Zenz, G., Zhou, X., Minack, E., Siberski, W., Nejdl, W.: From keywords to semantic queries - incremental query construction on the semantic web. Journal of Web Semantics **7**(3) (2009) 166–176