Caching and Prefetching Strategies for SPARQL Queries

Johannes Lorey
Hasso-Plattner-Institut
Potsdam, Germany
johannes.lorey@hpi.uni-potsdam.de

Felix Naumann
Hasso-Plattner-Institut
Potsdam, Germany
felix.naumann@hpi.uni-potsdam.de

ABSTRACT

Linked Data repositories offer a wealth of structured facts, useful for a wide array of application scenarios. However, retrieving this data using SPARQL queries yields a number of challenges, such as limited endpoint capabilities and availability, or high latency for connecting to it. To cope with these challenges, we argue that it is advantageous to cache data that is relevant for future information needs. However, instead of only retaining results of previously issued queries, we aim at retrieving data that is potentially interesting for subsequent requests in advance. To this end, we present different methods to modify the structure of a query so that the altered query can be used to retrieve additional related information. We evaluate these approaches by applying them to requests found in real-world SPARQL query logs.

1. INTRODUCTION

Linked Data sources offer a wealth of information about a multitude of topics, including geo-location facts[1], government data[2], or cross-domain information[3]. The SPARQL Protocol and RDF Query Language (SPARQL) has become the de facto query language to retrieve this information from publicly available endpoints.

However, whereas in principle SPARQL facilitates various information needs by defining complex query constructs, typical public SPARQL endpoint characteristics such as high latency, limited hardware resources, or unavailability restrict on-demand utilization of the data. In this work, we propose a novel approach for discovering and aggregating potentially relevant data for subsequent user requests based on previous query patterns by rewriting these preceding queries. To this end, we introduce some fundamental concepts of SPARQL in Sec. 1.1 and outline the contribution and organization of this paper in Sec. 1.2.

[1] `http://linkedgeodata.org`
[2] `http://data-gov.tw.rpi.edu/wiki`
[3] `http://dbpedia.org`

1.1 SPARQL Preliminaries

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 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 [10].

In terms of relational operations, `AND` represents an inner join of two graph patterns, `UNION` denotes their union, and `OPTIONAL` indicates a left outer join between P_1 and P_2. In addition, `AND` takes precedence over `UNION`, and `OPTIONAL` is always left-associative [10]. While `UNION` and `OPTIONAL` are reserved keywords in actual SPARQL queries to indicate the corresponding connection between two graph patterns, the `AND` keyword is omitted.

We call P the *parent graph pattern* of graph patterns $P_1, \ldots, P_n$, when it has the form $P = P_1 \oplus \ldots \oplus P_n$, where $\oplus$ is any one of the introduced keywords. Whereas $\oplus$ is symmetric for `AND` and `UNION`, it is not for `OPTIONAL` [10]. In any SPARQL query Q, the graph pattern P with no non-trivial parent graph pattern is referred to as the *query pattern* P_Q.

Typically, curly braces are used to delimit graph patterns: $\{P\}$. Here, if such a delimited graph pattern P represents a conjunction of multiple triple patterns T_i, i.e., its form is $P = T_1$ `AND` $T_2 \ldots$ `AND` T_n, we call P a *basic graph pattern* (BGP). While there is the notion of empty graph patterns in SPARQL, we only consider non-empty graph patterns.

In our work, we focus on `SELECT` queries. In general, SPARQL allows to limit the projection to certain variables listed after the `SELECT` statement. Moreover, graph patterns may contain so-called *filter conditions* indicated by the keyword `FILTER`. In a filter condition, specific restrictions, such as regular expressions or inequality relations, can be placed on resources to modify the scope of the selection of a query.

1.2 Contribution and Paper Organization

As with traditional Web search, SPARQL is suitable for different information needs, such as data exploration or navigational querying. However, in contrast to simple keyword-based queries, the well-defined structure and constructs of SPARQL queries allow for more fine-grained expressions representing the user intent. We build on this notion and present methods to modify queries in order to retrieve information relevant for subsequent related requests. Further, we discuss features of queries and query sequences influencing the suitability and effects of these modification methods.

By materializing results potentially relevant for subsequent queries locally, overall query execution time can be tremendously decreased for a client. Moreover, if the SPARQL endpoint becomes unavailable for a period of time, thus disallowing access to the remote information, a replicated subset of the data may be utilized instead. On the other hand, a SPARQL endpoint may also utilize knowledge about what data may be relevant for future queries, e.g., by storing it in main memory. The main trade-offs for this caching approach are increased storage requirements and potential data staleness which may be negligible given the concrete scenario.

The remainder of this paper is structured as follows: In Sec. 2, we discuss related work for this paper and point out differences in our approach. In Sec. 3, we introduce a means to group SPARQL queries by matching similarly structured requests. We then present our different query rewriting approaches in Sec. 4 and present an evaluation using real-world query logs in Sec. 5. Finally, we conclude this paper in Sec. 6 and highlight some future work in this context.

2. RELATED WORK

Related work for this paper draws mainly from two fields: (i) *semantic caching and prefetching*, e.g., techniques to retain previously fetched data or retrieve data relevant for subsequent queries, and (ii) *query relaxation*, e.g., parsing and modifying a user query to discover relevant resources.

2.1 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 major 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 either temporal or frequency aspects, e.g., by prioritizing least-recently or most-frequently used items. Such techniques also exist for SPARQL query result caching [12, 15]. Compared to these works, semantic caching employs more fine-grained information to characterize data, e.g., in order to establish variable-sized semantic regions containing related tuples [4] or detect data items with similar geolocation information [13].

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 or spatial features [6, 9]. However, there have been no attempts to prefetch RDF data based on the structure of sequential related SPARQL queries within and across query sessions.

2.2 Query Relaxation

Query relaxation aims at discovering interesting related information based on a user request. For keyword queries, this process is sometimes referred to as query expansion and has been a major research topic in the field of information retrieval (cf. [3]). In contrast to query refinement which aims at increasing precision by restricting the scope of a query, the goal of query relaxation or query expansion is to improve the recall in retrieval effectiveness. To this end, often precomputed metadata such as language models is utilized.

There exist a number of projects for implementing query relaxation for retrieving Linked Data. The authors of [8] suggest logical relaxations based on ontological metadata. In contrast, the approach in [7] relies on precomputed similarity tables for attribute values whereas in [5] the authors utilize a language model derived from the knowledge base.

In comparison, our rewriting strategies are not targeted at increasing recall when executing a single query, but instead aim to retrieve additional data related to future queries. Moreover, we do not assume any knowledge of the data source itself or of metadata describing it. Thus, while most previous approaches require at least some precomputation, our approach can be used ad-hoc solely by analyzing and modifying queries issued during run-time.

3. QUERY MATCHING

For different aspects of our work, we need to identify and cluster similarly-structured queries. To this end, we introduce our bottom-up query pattern matching approach based on matching similar triple patterns they contain.

3.1 Triple Pattern Distance

To match triple patterns, we determine their distance by accumulating the distance scores between their parts, i.e., between the two subjects, predicates, and objects, respectively. Here, if two triple pattern parts are both variables, their distance is 0. In case they are both URIs or both literals, their distance is the normalized Levenshtein distance of the respective strings. Otherwise, e.g., when one subject is a variable and the other a URI, the distance is 1. More formally, if x_1, x_2 are either the subjects, predicates, or objects of two triple patterns T_1, T_2, respectively, we define their symmetric distance score $\Delta(x_1, x_2)$ as

$$\Delta(x_1,x_2) := \begin{cases} 0, & \text{if } x_1 \in V \wedge x_2 \in V \\ \dfrac{levenshtein(x_1,x_2)}{max(|x_1|,|x_2|)}, & \text{if } (x_1 \in U \wedge x_2 \in U) \\ & \vee\ (x_1 \in L \wedge x_2 \in L) \\ 1, & \text{else.} \end{cases}$$

We determine the overall distance $\Delta(T_1, T_2) = \Delta(T_2, T_1)$ of two triple patterns T_1, T_2 by aggregating the individual triple pattern parts distance scores $\Delta(s_1, s_2)$, $\Delta(p_1, p_2)$, $\Delta(o_1, o_2)$ as follows: In case $\Delta(s_1, s_2) = \Delta(p_1, p_2) = \Delta(o_1, o_2) = 0$, we define $\Delta(T_1, T_2) := 0$. Otherwise, there exists a minimum triple pattern part distance score $min_\Delta := min(\Delta(s_1, s_2), \Delta(p_1, p_2), \Delta(o_1, o_2)))$ with $min_\Delta > 0$. In this case, the triple pattern distance score is defined as

$$\Delta(T_1,T_2) := \lceil \Delta(s_1,s_2) \rceil + \lceil \Delta(p_1,p_2) \rceil + \lceil \Delta(o_1,o_2) \rceil - (1 - min_\Delta)$$

This way, a distance $\Delta(T_1, T_2) \leq 1$ always indicates a dissimilarity in at most one triple pattern part, whereas for two non-equal triple pattern parts $1 < \Delta(T_1, T_2) \leq 2$, and a distance score $\Delta(T_1, T_2) > 2$ hints at differences between the two subjects, predicates, and objects.

Consider the two basic graph patterns BGP_1 and BGP_2 in Listing 1 and Listing 2, respectively, where the line numbers serve as identifiers for the included triple patterns. Here, the most similar triple pattern for T_1 in BGP_2 can be determined by computing $min(\Delta(T_1, T_4), \Delta(T_1, T_5), \Delta(T_1, T_6))$,

which results in $\Delta(T_1, T_5) = (\lceil 0 \rceil + \lceil 0 \rceil + \lceil \frac{12}{16} \rceil - \frac{4}{16}) = 0.75$. For T_2, the minimum value is $\Delta(T_2, T_6) = (\lceil 1 \rceil + \lceil 0 \rceil + \lceil 0 \rceil - 0) = 1$, and for T_3 it is $\Delta(T_3, T_4) = (\lceil 0 \rceil + \lceil \frac{5}{14} \rceil + \lceil \frac{5}{9} \rceil - \frac{9}{14}) \approx 1.36$. Thus, the most similar triple patterns for T_1, T_2, T_3 in BGP_2 are T_5, T_6, and T_4, respectively.

```
1 ?city1 rdfs:label "Paris"@fr .
2 ?person ?relationWith ?city1 .
3 :Auguste_Comte foaf:givenName "Auguste" .
```

Listing 1: Basic Graph Pattern Example BGP_1.

```
4 :Auguste_Comte foaf:surname "Comte" .
5 ?city2 rdfs:label "Montpellier"@en .
6 :Auguste_Comte ?association ?city2 .
```

Listing 2: Basic Graph Pattern Example BGP_2.

3.2 Basic Graph Pattern Matching

Using the triple pattern distance scores, we can now determine matchings between basic graph patterns. In our work, finding this matching is equivalent to deriving a perfect (complete) bipartite cover with minimum cost between the triple patterns of the two individual basic graph patterns where the cost is determined by the triple pattern distance $\Delta(T_i, T_j)$. Obviously, a perfect matching of triple patterns is only possible for a complete bipartite graph, i.e., for two BGPs with the same number of triple patterns. If this is not the case, in order to generate a biclique of triple patterns we pad the basic graph pattern containing less elements using dummy triple patterns T_ε so that for any triple pattern T the score $\Delta(T, T_\varepsilon) = \Delta(T_\varepsilon, T) = \infty$.

As optimal solutions for generating maximal matchings can be determined in polynomial time and the input size (i.e., the number of triple patterns in the two BGPs) is reasonably small, we choose the well-known Hungarian Method [11] to create an assignment with minimum cost. Furthermore, we assign a maximum cost threshold to all derived matchings of triple patterns. Here, we only consider triple pattern matchings with cost $\Delta(T_i, T_j) \leq 1$, i.e., all matched triple patterns are either identical or differ in at most one of non-variable subject, predicate, or object, respectively. The cost for triple pattern matchings with higher cost, i.e., matchings (T_i, T_j) with $\Delta(T_i, T_j) > 1$ is set to ∞.

The score $\Delta(BGP_1, BGP_2)$ of derived complete matchings $M_T \subset \{(T_i, T_j) | (T_i, T_j) \in BGP_1 \times BGP_2\}$ is defined as:

$$\Delta(BGP_1, BGP_2) := \frac{\sum_{(T_i,T_j)\in M_T} \Delta(T_i, T_j)}{|M_T|}$$

If the result of the Hungarian method for M_T contains any individual triple pattern matching with infinite cost, $\Delta(BGP_1, BGP_2)$ is also infinite. In this case, no complete matching with only finite triple pattern distance scores can be determined between BGP_1 and BGP_2. Assuming such a complete matching containing only valid triple pattern matchings exists, this BGP matching would have been the result of the algorithm as its cost would be $< \infty$.

Conversely, if the algorithm determines an optimal matching with infinite cost, any other matching with cost $< \infty$ cannot be complete as the algorithm does not terminate before discovering a maximal matching. Given our setting of a complete bipartite graph (or biclique), any maximal matching is always bound to be a complete matching.

Applying this approach on the basic graph patterns illustrated in Listing 1 and Listing 2 to determine a complete matching with minimum cost yields the same triple pattern matchings listed earlier. Thus, the optimal matching $\{(T_1, T_5), (T_2, T_6), (T_3, T_4)\}$ has cost $\frac{0.75+1+\infty}{3}$, i.e., BGP_1 and BGP_2 cannot be matched to another.

3.3 Query Pattern Matching

Real-world SPARQL query patterns can be more complex than simple basic graph patterns, i.e., they may contain multiple recursively layered BGPs connected using the `UNION` or `OPTIONAL` keyword as introduced in Sec. 1.1. Along the lines of the previous subsection, where we defined BGP matching using the contained triple patterns, our matching approach for general query patterns is based on recursively matching the (basic) graph patterns they consist of.

Due to the recursive structure of SPARQL queries Q_1, Q_2, we can only try to match two basic graph patterns BGP_i, BGP_j to one another if these BGPs reside at the same recursion level of the query patterns P_{Q_1}, P_{Q_2}, respectively. Additionally, the two basic graph patterns and any of their parent graph patterns also need to be connected by the same keyword to other (basic) graph patterns at their respective recursion level. If these two conditions are met, we say that BGP_i, BGP_j can be *aligned* to each other. More generally, if all (basic) graph patterns contained in a parent graph pattern P_1 can be aligned to at least one (basic) graph pattern of another parent graph pattern P_2 and vice versa, P_1, P_2 can also be aligned to each other.

Consider the three SPARQL parent graph patterns

$$P_1 := BGP_1 \texttt{ OPTIONAL } (BGP_2 \texttt{ UNION } BGP_3)$$
$$P_2 := BGP_4 \texttt{ OPTIONAL } (BGP_5 \texttt{ UNION } BGP_6),$$
$$P_3 := BGP_7 \texttt{ OPTIONAL } BGP_8,$$

where $BGP_1, \ldots, BGP_8$ are basic graph patterns. Here, BGP_1 can be aligned to BGP_4 or BGP_7. Additionally BGP_2 or BGP_3 can be aligned to either BGP_5 and BGP_6, respectively. However, BGP_8 cannot be aligned to any other basic graph pattern in P_1 or P_2, because its recursion level is different from those of $BGP_2, BGP_3, BGP_5, BGP_6$ and `OPTIONAL` is not symmetric, i.e., BGP_7 `OPTIONAL` $BGP_8 \neq BGP_8$ `OPTIONAL` BGP_7. Thus, while P_1 and P_2 can be aligned to each other, P_3 cannot be aligned to either of them.

We use a bottom-up approach to try to match two query patterns P_{Q_1}, P_{Q_2} to one another. Similarly to the matching method introduced in the previous subsection, we try to derive a complete matching with minimal (finite) cost between all basic graph patterns of P_{Q_1} and P_{Q_2} that can be aligned to each other. Here, the cost between two basic graph patterns is determined by their distance score $\Delta(BGP_1, BGP_2)$.

If for any BGP in P_{Q_1} or P_{Q_2} no alignment is possible or its matching has infinite cost, P_{Q_1} or P_{Q_2} cannot be matched. If we derive a complete matching for all basic graph patterns, we check whether the parent graph patterns of all pairs of matched BGPs can be aligned to each other. If this is the case, we continue checking the alignment of the parent graph patterns until we reach the query pattern.

In case no alignment is possible, the two query patterns P_{Q_1} and P_{Q_2} cannot be matched. Conversely, two query patterns P_{Q_1} and P_{Q_2} can be matched if they can be aligned to each other and a complete matching with finite cost can be established between all basic graph patterns BGP_i and BGP_j in P_{Q_1} and P_{Q_2}, respectively. Canonically, any graph pattern can always be matched to itself.

To group structurally similar queries, we introduce the notion of *query clusters*. All queries in a query cluster can be matched to all other queries within the cluster, i.e., there exists a pairwise complete matching with finite cost between all BGPs of any two queries in the same cluster. Note that query clusters may be overlapping, i.e., a query can be element of multiple query clusters.

4. QUERY AUGMENTATION

The main motivation of our work lies in retrieving information for SPARQL queries that is also relevant for subsequent related requests. Beyond basic caching, we argue for prefetching results: Here, we attempt to modify a query to retrieve additional data potentially relevant for future information needs. In this section, we motivate and illustrate different approaches for modifying queries accordingly.

4.1 Augmentation Concepts

We call the process of modifying the query contents *query augmentation* to emphasize that the results retrieved by issuing the original query are included in the result set for the modified query. In other words, the matches for the unmodified query form a subgraph of the matches for the augmented query. In this work, we are typically interested in retrieving additional information related to a *central concept*, namely the subject (i.e., either a variable or URI) occurring most often in the query pattern. Moreover, we require a central concept to be either part of the projection if it is a variable or to influence the selection if it is a URI. We assume that a URI influences the selection if at least one triple pattern, in which the URI is the subject, contains a projection variable.

Our intuition of query augmentation builds on concepts from information retrieval. For example, in traditional keyword-based search engines, a user might be unaware of the most suitable string pattern to enter to retrieve all relevant results at once. However, in several iterations the user may choose to refine the initial query based on retrieved results. In Linked Data terms, a user might query for more detailed information about a certain resource or for similar information of related resources after analyzing preliminary results, thus incrementally modifying the initial query.

In the remainder of this section, we introduce the different augmentation strategies we have implemented while emphasizing the particular requirements for their application. We illustrate the effect of applying these strategies on Query 1. Put simply, for the central concept French philosopher `:Auguste_Comte` this query retrieves the birth place(s) located in France alongside all influences on him that died in Paris. Table 1 lists all bindings retrieved by issuing Query 1 against the public DBpedia SPARQL endpoint[4] currently containing DBpedia 3.8 data.

```
PREFIX    : <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?birthPlace ?influence WHERE {
    :Auguste_Comte dbo:birthPlace ?birthPlace .
    ?birthPlace dbo:country :France .
    :Auguste_Comte dbo:influencedBy ?influence .
    ?influence dbo:deathPlace :Paris .
}
```

Query 1: Example of a SPARQL query

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

birthPlace	influence
:Montpellier	:Jean-Baptiste_Say
:Montpellier	:Émile_Durkheim

Table 1: All results for Query 1 in DBpedia 3.8

4.2 Exploratory Augmentation

In *exploratory augmentation*, we query for additional facts that are available for the central concept. The idea of exploratory augmentation is that based on previous results a user might be interested in more information for a specific resource. However, this might also be the case if the initial result set is empty, e.g., because of misspelled or ambiguous vocabulary terms (such as `foaf:img` and `foaf:Image`).

Potentially, there may also exist certain divergences between ontological information assumed by the user and the vocabulary used in actual data. For example, we discovered that although a number of properties are used frequently for instances of certain types in DBpedia, they are not defined in the ontology (e.g., `dbo:anthem` for `dbo:Country`). On the other hand, there are also a number of defined properties that are rarely used in instance data for the corresponding classes (e.g., `dbo:depth` for `dbo:Place`) [1].

This augmentation is applied by adding a triple pattern to the query, where the subject is the central concept and predicate as well as object are unique variables. Moreover, these two variables are added to the projection, thus evaluating all facts in which the central concept is subject. We highlight the corresponding changes in the resulting Query 2 by underlining modified or added triple patterns and variables.

```
SELECT ?p ?o ?birthPlace ?influence WHERE {
    :Auguste_Comte dbo:birthPlace ?birthPlace .
    ?birthPlace dbo:country :France .
    :Auguste_Comte dbo:influencedBy ?influence .
    ?influence dbo:deathPlace :Paris .
    :Auguste_Comte ?p ?o .
}
```

Query 2: Exploratory augmentation of Query 1

An excerpt of the extended result set for Query 2 is listed in Tab. 2. Please note that we exclude the bindings listed in Tab. 1 which are also contained in the result set of Query 2.

p	o
rdf:type	dbo:Person
dbo:birthDate	1798-01-19
dbo:notableIdea	:Positivism
dbo:influenced	:Karl_Marx
...	

Table 2: Some results for Query 2 in DBpedia 3.8

4.3 Template Augmentation

For the *template augmentation* approach, we first need to identify the cluster the current query belongs to as discussed in Sec. 3.3 by matching the current query pattern to those of preceding requests. If we determine a matching with finite cost, we require this matching to be non-trivial, i.e., to include at least one match between two non-identical basic graph patterns. This is true if the overall distance score of the query pattern matching is greater than 0, i.e., if any two matched basic graph patterns contain a pair of triple patterns (T_i, T_j) for which either the subjects, predicates, or objects differ.

A matching between two query patterns P_1 and P_2 meeting these requirements allows us to construct a *query template*: Initially, the template query string is identical to the current query string. We then introduce unique variables to replace all differing triple pattern parts included in the matching, i.e., either the non-matching subject, predicate, or object of triple patterns (T_i, T_j) for which $0 < \Delta(T_i, T_j) \leq 1$. Here, if the same resource in a BGP is replaced more than once, we use the same unique variable to ensure consistency.

Instead of issuing many similarly structured queries with only little variance, e.g., by using a crawler, a query template instead retrieves all relevant information using only a single query. A possible query template for Query 1 is illustrated in Query 3. Here, one specific resource has been replaced by a variable, which has also been added to the projection of the query in the `SELECT` statement. As indicated in Tab. 3, the result set contains the previous bindings as well as information about other persons with similar properties, e.g., about poet Paul Fort or scientist Pierre de Fermat.

```
SELECT ?s ?birthPlace ?influence WHERE {
    ?s dbo:birthPlace ?birthPlace .
    ?birthPlace dbo:country :France .
    ?s dbo:influencedBy ?influence .
    ?influence dbo:deathPlace :Paris .
}
```

Query 3: Template augmentation of Query 1

s	birthPlace	influence
:Auguste_Comte	:Montpellier	:Jean-Baptiste_Say
:Auguste_Comte	:Montpellier	:Émile_Durkheim
:Paul_Fort	:Reims	:Paul_Verlaine
:Pierre_de_Fermat	:Beaumont-de-Lomagne	:François_Viète
...		

Table 3: Some results for Query 3 in DBpedia 3.8

4.4 Type Augmentation

If class membership information in the knowledge base is available for the central concept, exploiting this ontological data can help in discovering information for related resources. In *type augmentation* we identify the `rdf:type` of the central concept and retrieve data for the instances belonging to the same classes by replacing the central concept with a unique variable throughout the query.

The goal of type augmentation is similar to that of template augmentation, i.e., querying information about different related resources. Whereas in template augmentation this relatedness is solely determined by the context of the replaced triple pattern part, in type augmentation it is derived by exploiting both structural and ontological information. Assuming the central concept is identical to the triple pattern part replaced for matching BGPs, all bindings retrieved using type augmentation are also retrieved using template augmentation. However, for type augmentation the connection between different resources may be stronger than for template augmentation given their type information.

According to the RDF Schema[5], a resource may be instance of multiple classes, where these classes may either be unrelated or reside at different levels of the same type hierarchy. Therefore, RDF resources may be instances of very generic types, such as `owl:Thing`. Hence, one challenge for type augmentation lies in determining a suitable class for which instance data is retrieved, especially without assuming any a priori knowledge of the underlying ontology.

A number of techniques can be employed to gather this data, e.g., using multiple preliminary queries to construct a simple type hierarchy or utilizing aggregate functions such as `COUNT` to generate heuristics about the distribution of different types. In our approach, we introduce a `FILTER NOT EXISTS` to exclude all those (generic) types that have (more specific) subclasses. By doing so, we assume that the endpoint supports SPARQL 1.1 expressions and all resources are instances of at least one leaf node in the type hierarchy. If this is not the case, we exclude the filter condition.

Query 3 illustrates the result of applying type augmentation on the reference query, i.e., by introducing the new triple patterns regarding `rdf:type` information, exchanging the central concept in all other triple patterns, and applying the filter condition. Whereas the results for template augmentation also include bindings for `?s` that are instances of rather generic classes such as `dbo:Person`, the results for Query 3 listed in Tab. 4 only include resources that are instances of specific subclasses, e.g., `dbo:Philosopher`.

```
SELECT ?s ?birthPlace ?influence WHERE {
    :Auguste_Comte rdf:type ?type .
    ?s rdf:type ?type .
    ?s dbo:birthPlace ?birthPlace .
    ?birthPlace dbo:country :France .
    ?s dbo:influencedBy ?influence .
    ?influence dbo:deathPlace :Paris .
    FILTER NOT EXISTS {?t1 rdfs:subClassOf ?type}
}
```

Query 4: Template augmentation of Query 1

s	birthPlace	influence
:Auguste_Comte	:Montpellier	:Jean-Baptiste_Say
:Auguste_Comte	:Montpellier	:Émile_Durkheim
:Jean-Paul_Sartre	:Paris	:Maurice_Merleau-Ponty
:René_Descartes	:Descartes,_Indre-et-Loire	:Marine_Mersenne
...		

Table 4: Some results for Query 4 in DBpedia 3.8

4.5 Holistic Augmentation

The intuition of *holistic augmentation* is that the scope of SPARQL queries can be broadened by removing certain triple patterns they contain. However, to ensure that the result set of this modified query still contains all results of the original one, the removed parts must not contain variables essential to the projection or selection of a query. In other words, the variables in the `SELECT` statement still need to be present in the modified query so that they may be bound to an RDF term in a graph matching.

Typically, if we select a triple pattern to remove from the basic graph pattern BGP_i that contains it, we also remove the same triple pattern from any other basic graph pattern BGP_j that can be matched to BGP_i as described in Sec. 3.3. We call such a removal *valid*, if applying it on a valid SPARQL

[5] `http://www.w3.org/TR/rdf-schema`

query results in a valid SPARQL query, i.e., if all projection variables are referenced in at least one remaining triple pattern of the query pattern. Note that there exist queries for which no valid triple pattern removal is possible, e.g., queries containing only one triple pattern.

To identify the most suitable triple pattern to remove from a query, we utilize the *variable counting* heuristic introduced in [14]. Essentially, this heuristic is based on the assumption that unbound subjects are more selective than unbound objects which in turn are more selective than unbound predicates. Hence, generally a triple pattern with an unbound predicate, but bound subject and object matches less RDF statements in a knowledge base than a triple pattern with either only an unbound subject or object, i.e., is more selective. Also, a triple pattern with two or three unbound parts is less selective than a triple pattern with only one or two unbound parts, respectively. Thus, the least selective triple pattern is the one containing only variables.

In any query pattern there can be more than one triple pattern with maximum selectivity. In this case, we select an arbitrary one for removal. If this removal is not valid, we check whether a valid removal can be achieved for a different triple pattern with same or lesser selectivity. We continue until we have either exhaustively checked all triple patterns or discovered a validly removable triple pattern. In the latter case, we modify the query by deleting the triple pattern from the parent basic graph pattern and any other basic graph pattern this BGP can be matched to.

Removing a highly-selective query triple pattern not essential for the projection mostly assists in situations where (i) the query is too restrictive, (ii) the query contains invalid statements, or (iii) the data or ontology in the knowledge base is inconsistent, e.g., as described in [1]. One of the two most selective triple pattern in Query 1 is crossed out in Query 5, thereby indicating its removal in this augmented query. Table 5 lists some results for Query 5.

```
SELECT ?birthPlace ?influence WHERE {
    :Auguste_Comte dbo:birthPlace ?birthPlace .
    ?birthPlace dbo:country :France .
    :Auguste_Comte dbo:influencedBy ?influence .
    ~~?influence dbo:deathPlace :Paris .~~
}
```

Query 5: Holistic Augmentation of Query 1

birthPlace	influence
:Montpellier	:Adam_Smith
:Montpellier	:David_Hume
:Montpellier	:Jean-Baptiste_Say
:Montpellier	:Émile_Durkheim
...	

Table 5: Some results for Query 5 in DBpedia 3.8.

5. EVALUATION

To evaluate the applicability of our prefetching strategies for SPARQL queries, we analyzed parts of the DBpedia 3.6 and LinkedGeoData (LGD) query logs provided for the USEWOD 2013 data challenge [2]. The log files contain a number of requests received by the respective public SPARQL endpoints and were collected for different dates in 2011. While the specific metadata provided in the Apache Common Log Format differs slightly for the individual services, all requests contain the sender's anonymized IP address and a timestamp in addition to the actual query.

5.1 Methodology

We use the timestamp information to provide a meaningful segmentation for successive queries by introducing *query sessions*. A query session is a chronologically ordered sequence of at least two queries issued sequentially by the same user (represented by a unique IP address) over a period of time. We call query sessions *homogeneous*, if they contain queries belonging to the same cluster, i.e., any query in a session can be matched to any other query in the same session. Otherwise, we call this query session *heterogeneous*.

We exploit statistical features and metadata extracted from the query logs format for delimiting a query session. Specifically, we restrict the duration of a session by comparing the timestamps of issued queries with that of the initial query of a session. Once this difference exceeds a certain threshold, we assume a new query session has started. Depending on the analyzed dataset, we put an upper limit on the number of queries a single session may contain, thereby splitting a single query sequence into separate sessions if the number of successive queries is greater than this limit.

Matchings for triple patterns can easily be transformed into triples by applying the corresponding variable bindings. By materializing these triples for the original, unmodified query and for all queries generated by applying the augmentation strategies introduced in Sec. 4, we maintain individual result set caches. We then evaluate how many triples generated for bindings of subsequent queries are already contained in the individual caches. If we determine that a newly generated triple is contained in a cache, we consider this a *cache hit*. Note that the set of cached triples for each augmentation strategy includes at least the cached triples for the original query.

We use OpenLink Virtuoso Open-Source Edition version 6.1.3 as SPARQL endpoint containing the English DBpedia 3.6 dataset released on January 17, 2011, and the LinkedGeoData dump released on April 26, 2011 in separate named graphs. For each augmentation strategy, we restrict the number of retrieved results for performance reasons to a maximum of 100,000 using the `LIMIT` keyword for SPARQL.

5.2 DBpedia 3.6 Query Logs

The requests included in the DBpedia 3.6 query logs exhibit timestamps in hourly resolution, e.g., `[24/Jan/2011 01:00:00 +0100]`. In our analysis, only successive queries from the same user with identical timestamps may belong to the same session. However, as this heuristic introduces some vagueness regarding the contents of a session, e.g., by ignoring session timeouts, we also limit the maximum number of queries belonging to any single session to 25. Notice that by applying this conservative restriction, we potentially lower our number of cache hits which will most likely increase for longer sessions. In summary, once a user query with a different timestamp is detected or we discover more than 25 successive queries, we assume a new session has started.

Consequently, we base our evaluation on 288 query sessions for which we were able to retrieve results for at least one contained query. Of these query sessions, 176 (61%) were homogeneous. On average, the query sessions contained around 21 queries. In around 34% of all query sessions, we could not identify any cache hits. We assume that

this is because the total result set size for these sessions is relatively small: The sessions with no cache hits only result in about 100 generated triples compared to around 3,300 triples for sessions with cache hits. There are two possible reasons for this: (i) Our local SPARQL endpoint did not contain the entire DBpedia 3.6 corpus and (ii) some queries did not yield any results even when executed against the public DBpedia endpoint (e.g., because of syntax errors).

For all sessions with at least one cache hit, we illustrate the total number of cache hits in relation to the total number of generated (unique) triples for all unmodified queries in a session in Fig. 1. Each marker represents the best augmentation strategy resulting in the most cache hits for this session. If for none of the augmentation strategies the number of hits was greater than the cache hits of the original query, the marker "no augmentation" is used.

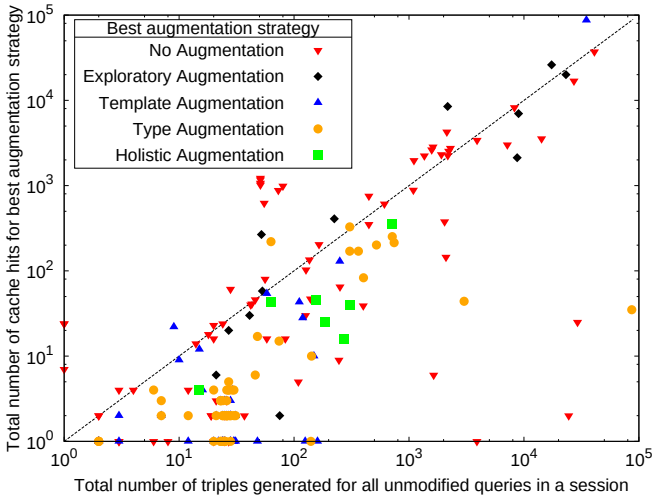

Figure 1: Best augmentation strategy when caching results of the first query in a session in the DBpedia 3.6 log files

Overall, our findings indicate that caching the results of the first, unaugmented query of a session yields the most amount of cache hits in about 38% of all analyzed query sessions. The results for type and template augmentation have the most cache hits in 28% and 23% of the sessions, respectively, whereas applying exploratory and holistic augmentation on the first session query only results in the most cache hits for 7% and 4% of all sessions.

When considering homogeneous query sessions only, type augmentation yields the most cache hits in 39% of all sessions, followed by no augmentation (30%), template augmentation (28%), exploratory and holistic augmentation (1% each). Notice that we discovered a number of homogeneous query sessions which contain the exact same query multiple times. For example, this is the case if only one triple is generated for all queries in a session and identified as cache hit repeatedly as represented by markers located on the y-axis. Obviously, if the exact same query is issued over again within the course of a session, no additional cache hits can be generated when applying an augmentation strategy.

The mean number of cache hits is highest for exploratory augmentation (4,541 cache hits) and lowest for type augmentation (33 cache hits) when considering only those sessions where these two augmentation strategies yield the most cache hits. On average, in all sessions with markers above the diagonal (indicated by the dashed line in Fig. 1), each generated triple represents a cache hit at least once.

We also evaluated how caching the result of every (augmented) query influenced the number of cache hits for subsequent (unmodified) queries in a session. While the number of query sessions with no cache hits dropped to around 24%, for those query sessions with cache hits we observed comparable results to the ones illustrated. Hence, we assume that by analyzing the first query, a suitable caching strategy can be determined for all subsequent queries of the same session. For example, for homogeneous query sessions applying template or type augmentation on the first query most likely results in the same augmented query as applying it on any of the subsequent session requests.

In general, due to the large number of homogeneous query sessions in the DBpedia query logs, type and template augmentation appear to be the most successful among the augmentation strategies. Given the large number of resources in DBpedia and our restriction on the maximum number of results, we are impairing the success of these strategies to some extent, as many potentially relevant facts are simply not retrieved. Without this restriction, these prefetching strategies should yield even better results.

On the other hand, the amount of query sessions benefiting most from holistic or exploratory augmentation is limited. This might stem from the apparently small number of queries issued by human users, towards which these strategies are targeted. Moreover, in more than half the query sessions (55%) holistic augmentation could not be applied as no valid triple pattern removal was possible. Naturally, in these cases the number of cache hits for holistic augmentation equals the one of not applying any augmentation.

5.3 LinkedGeoData Query Logs

As the timestamps provided for the queries in the LGD logs are precise to the second, we delimit query sessions in these logs more accurately by introducing a session timeout and maximum session duration. If for a query from a specific user we cannot discover another query from the same user within 10 minutes time, we assume the identified query is the last one in a session. Overall, we delimit a query session by restricting its duration to a maximum of 60 minutes, its session timeout to 10 minutes, and its maximum number of queries to 50 (whichever comes first).

As with the DBpedia query logs, we analyzed only those query sessions in which we determined at least one query for which we were able to generate a result as described above, i.e., we based our evaluation on 440 query sessions. This time, for only 9% of these query sessions no cache hits could be discovered at all. The analysis of which augmentation strategy resulted in the most cache hits for the remaining 424 query sessions is illustrated in Fig. 2.

For the LGD logs, caching the results of unmodified queries resulted in the most hits (48% of all query sessions), followed by exploratory augmentation (26%), template augmentation (15%), type and holistic augmentation (5% each). For heterogeneous query sessions (55% of all query sessions), exploratory augmentation is the best augmentation strategy (30%), followed by template augmentation (19%), holistic augmentation (7%), and type augmentation (6%).

We also discovered that the number of mean cache hits was much higher than the ones of the DBpedia log files: For those sessions that benefited from template augmentation

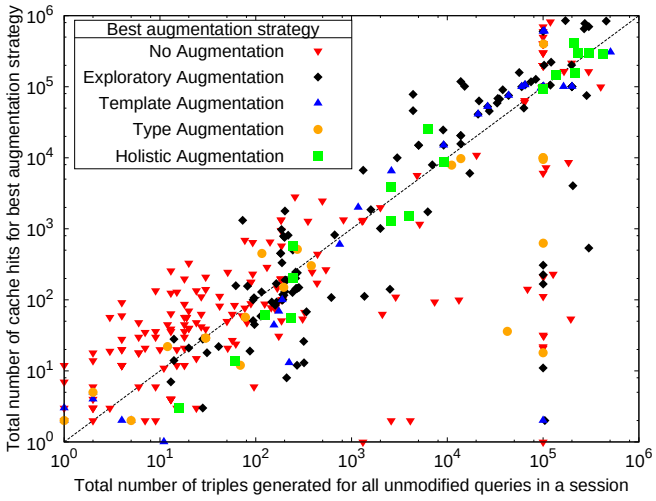

Figure 2: Best augmentation strategy when caching results of the first query in a session in the LinkedGeoData log files

the average number of cache hits was highest with 72,917 and lowest for type augmentation with 48,320. On the other hand, the average session length was comparatively small with only 12 queries per session. This could be because the LGD log queries were actually issued by human users (as opposed to crawlers or other software agents). Intuitively, this would also explain why the majority of query sessions are heterogeneous: Whereas software agents use somewhat hard-coded HTTP requests to retrieve Linked Data, human users are more flexible when issuing queries, e.g., by using the LGD SPARQL web interface[6].

Again, caching the result of every query in a session had only little impact on the choice of augmentation strategy, which is to be expected considering the small mean number of queries in a session. However, the percentage of query sessions not benefiting from any caching decreased slightly to 6%. In general, the cached results for the queries can almost always be used for subsequent queries in a session for the LGD logs. Again, our local SPARQL endpoint might have not contained all data available in the public SPARQL endpoint. However, the impact on our results is negligible as we were able to generate query results for the vast majority of sessions (75%) and cache hits in almost all of these (91%).

6. CONCLUSION AND OUTLOOK

In this paper, we have presented a number of approaches to modify SPARQL queries with the goal to retrieve additional results that are potentially relevant for subsequent requests of the same query session. We evaluated how often and how many of these additional results are actually relevant by identifying query sessions that can exploit this prefetched data in subsequent requests compared to only caching the results of the unmodified query.

While the majority of all analyzed query sessions benefited from caching the results of the unmodified or augmented first query (around 66% for DBpedia and 91% for LinkedGeoData), the most suitable augmentation strategy (if any) differs from session to session. For example, large-scale homogeneous query sessions can exploit a cache containing similar data of related resources, e.g., as generated by applying template and type augmentation.

On the other hand, human users might retrieve more "diverse" information, e.g., specific facts about a resource as generated by applying exploratory augmentation. Overall, whereas not all augmentation strategies can be applied on certain queries (e.g., holistic augmentation for queries containing only one triple pattern), combining different strategies during the course of a query session might still prove advantageous for such diversified scenarios.

In future work, we aim at capturing the intent in a query session more accurately, e.g., by using more fine-grained approaches than that of a central concept. Based on this, more customized augmentation strategies are conceivable, e.g., by analyzing individual resources contained in a query. Finally, we want to be able to discover on the most suitable augmentation strategy based on the contents of a query session and thus adjust other query sessions accordingly.

[6] http://linkedgeodata.org/sparql

7. REFERENCES

[1] Z. Abedjan, J. Lorey, and F. Naumann. Reconciling ontologies and the web of data. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1532–1536, Maui, HI, USA, October 2012.

[2] B. Berendt, L. Hollink, M. Luczak-Rösch, K. H. Möller, and D. Vallet. USEWOD2013 – 3rd international workshop on usage analysis and the web of data. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, Montpellier, France, 2013.

[3] C. Carpineto and G. Romano. A survey of automatic query expansion in information retrieval. *ACM Comput. Surv.*, 44(1):1:1–1:50, Jan. 2012.

[4] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 330–341, Bombay, India, 1996.

[5] S. Elbassuoni, M. Ramanath, and G. Weikum. Query relaxation for entity-relationship search. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 62–76, Heraklion, Greece, 2011.

[6] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Transactions on Information Systems*, 24(1):51–78, 2006.

[7] A. Hogan, M. Mellotte, G. Powell, and D. Stampouli. Towards fuzzy query-relaxation for RDF. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 687–702, Heraklion, Greece, 2012.

[8] C. A. Hurtado, A. Poulovassilis, and P. T. Wood. Journal on data semantics X. chapter Query relaxation in RDF, pages 31–61. Springer-Verlag, Berlin, Heidelberg, 2008.

[9] S. Jonassen, B. B. Cambazoglu, and F. Silvestri. Prefetching query results and its impact on search engines. In *Proceedings of the ACM International Conference on Information Retrieval (SIGIR)*, pages 631–640, Portland, OR, USA, 2012.

[10] C. G. Jorge Pérez, Marcelo Arenas. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):16:1–16:45, 2009.

[11] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logist. Quarterly*, 2(1-2):83–97, 1955.

[12] M. Martin, J. Unbehauen, and S. Auer. Improving the performance of semantic web applications with SPARQL query caching. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pages 304–318, Heraklion, Greece, 2010.

[13] Q. Ren and M. H. Dunham. Using semantic caching to manage location dependent data in mobile computing. In *Proceedings of the International Conference on Mobile Computing and Networking*, pages 210–221, Boston, MA, United States, 2000.

[14] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 595–604, New York, NY, USA, Apr. 2008.

[15] M. Yang and G. Wu. Caching intermediate result of SPARQL queries. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 159–160, Hyderabad, India, 2011.