Performance Improvement of Triple Stores via Concept-Based Caching

Claus Stadler  
Institute of Computer Science, University of Leipzig  
buehmann@informatik.uni-leipzig.de

René Speck  
InfAI - Institute for Applied Informatics e.V.  
speck@informatik.uni-leipzig.de

Michael Martin  
Institute of Computer Science, University of Leipzig  
martin@informatik.uni-leipzig.de

Axel-Cyrille Ngonga Ngomo  
Institute of Computer Science, University of Leipzig  
ngonga@informatik.uni-leipzig.de

Jens Lehmann  
Institute of Computer Science, University of Leipzig  
lehmann@informatik.uni-leipzig.de

ABSTRACT
SPARQL is a W3C RDF standard that specifies the primary query language for access to RDF datasets. As a consequence, the performance of executing SPARQL queries is crucial for applications, which manipulate or present the information stored in RDF triple stores. In this paper, we propose a caching mechanism, which improves SPARQL query performance of triple stores under typical workloads. The mechanism identifies relevant patterns and, in contrast to previous work, supports a larger fragment of SPARQL, in particular FILTER constraints. Moreover, the cache size can be regulated via graph caching algorithms, which control the selective invalidation and creation of cached result sets. We evaluated our approach using a generic RDF benchmarking framework applied to the DBpedia dataset and obtained significant performance improvements over previous work.

Keywords: caching, SPARQL, RDF, clustering

1. INTRODUCTION
As a practical example, the LinkeGeoData, an RDF version of the OpenStreetMap dataset, contains worldwide features, such as bakeries. Consider a query for bakeries in Leipzig. There are multiple ten-thousands bakeries worldwide, and so there are features in Leipzig. Hence, although the result set of that query is small (around 150), an RDF store has to scan multiple ten-thousands of resources. Faceted browsing applications that create conjunctive queries by incrementally adding constraints may benefit from such a cache as prior result sets can be readily re-used.

2. PRELIMINARIES

Definition 1 (Generalized RDF Graph). In accordance with the RDF specification, let $\mathcal{I}$, $\mathcal{B}$, $\mathcal{L}$ be pairwise disjoint sets of IRIs, blank nodes and literals, respectively. The set of RDF Terms is then defined as $T := \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$. The generalized RDF graph is defined as $G := T \times T \times T$.

Definition 2 (Triple Pattern). A triple pattern is a member of the set $((T \cup V) \times (I \cup V) \times (T \cup V))^2$.

2.1 SPARQL Algebra

Definition 3 (SPARQL Dataset). The active graph is the graph from the dataset used for basic graph pattern matching.

Definition 4 (Active graph).

Definition 5 (Basic Graph Pattern).

Definition 6 (SPARQL Query Syntax and Semantics). Basic Graph Pattern ($BGP$):
- If $P_1$ and $P_2$ are graph patterns, then so are $(P_1 \text{ AND } P_2)$, $(P_1 \text{ OPT } P_2)$ and $(P_1 \text{ UNION } P_2)$.
- $[BGP]$
unordered collection of elements in which each element may appear more than once. It is described by a set of elements and a cardinality function giving the number of occurrences of each element from the set in the multiset.

### 3. PROBLEM STATEMENT

#### completeness, correctness

In general, for the evaluation of (SPARQL) algebra expressions, the Frege principle holds: The result of an evaluation of an whole is a function over the evaluation of its parts:

\[ \phi_O([a_1, \ldots, a_n]) = \phi_O([a_1], \ldots, [a_n]) \]

whereas \( \phi_O \) is the semantic composition function corresponding to \( O \).

Let \( SAE \) be the set of SPARQL algebra expressions. Further, a \textit{generalized SPARQL cache} \( GSC \) is a partial function \( SAE \rightarrow \mathcal{P}(\Omega) \) that associates a \( SAE \) with a set of solution bindings. Further, let \( \text{cost} : SAE \rightarrow \mathbb{R} \) a function that assigns an estimated execution cost to \( SAEs \).

Query containment is defined as:

\[
A \subseteq B \iff \{ t \mid t_A \in [A]_D \} \subseteq \{ t_B \mid t_B \in [B]_D \} \text{ for every } D
\]

Now, the problems in increasing complexity are:

- Given two \( SAEs \) \( q \) and \( c \), that act as the query and cache, respectively, determining where there exists a mapping of the variables \( \mu : V \rightarrow V \) such when applied to the cache \( SAE \), it is subsumed by the query \( \mu(c) \subseteq q \). The function \( \text{applyCache} : (q : SAE, \mu : VMc : SAE, s : SAE) \rightarrow SAE \) function takes as input a query, cache, and substitution \( SAE \), and yields a new \( q' \).

- The next step is about choosing from a set of caching candidates, with the goal of finding a sequence of \( \text{applyCache} \) applications, such that the overall cost of the query is minimized.

Besides the problems related to SPARQL algebra transformations, there are further issues that need to be addressed:

- Cache selection: Determine whether a query is worth caching.
- Cache eviction: Determine which cache entries to remove.
- Join order optimization: As the cache and resides in a different database system than the original data, the caching system must be capable to perform joins.

### 4. APPROACH

#### 4.1 The simple case

We need an operator that represents a reference to a cache entry

In this section, we only consider basic graph patterns and filters.

A basic \textit{SPARQL cache} is a partial function that associates a basic graph pattern with a set of corresponding solution bindings \( c : BPG \rightarrow \Omega \). The following constraints should hold: \( \text{vars}(BPG) \subseteq \text{vars}(\Omega) \), all solution bindings should be compatible with \( \Omega \). Given a SPARQL query \( Q \), the goal is to substitute its graph patterns such that the cache is made use of.

#### 4.2 Extending with filters

Given a Query BGP:

\(?s \ a \ Bakery\)

\(?s \ locatedIn \ Leipzig\)

it is possible to normalize it by replacing

\(?a \ ?b \ ?c\).

\(?a \ ?d \ ?e\)

\(\text{FILTER(}\ ?b = \text{rdf:}\type \&\& \ ?c = \text{Bakery} \&\& \ ?d = \text{locatedIn} \&\& \ ?e = \text{Leipzig})\)

By converting the filter expression to a CNF:

\[\{\{?b = \text{rdf:}\type\}, \{?c = \text{Bakery}\},\{?d = \text{locatedIn}\}, \{?e = \text{Leipzig}\}\} \]

and adding the triples, we obtain a set-based unified representation:

\[\{\{\text{triple(?a, ?b, ?c)}\}, \{\text{triple(?a, ?d, ?e)}\}\},\{?b = \text{rdf:}\type\}, \{?c = \text{Bakery}\},\{?d = \text{locatedIn}\}, \{?e = \text{Leipzig}\}\} \]

By this representation, relaxation of queries can be simply done by omitting clauses.

Now let’s assume there is a cache:

\[\{\{\text{triple(?x, ?y, ?z)}\}, \{?y = \text{locatedIn}\}, \{?z = \text{Leipzig}\}\} \rightarrow C1\]

Then there exists a mapping \( \mu : \{\text{?x} \rightarrow \text{?a, ?y} \rightarrow \text{?y}, \text{?z} \rightarrow ?e\} \). In practice, the mapping can be efficiently computed by grouping the expressions by a blocking key (equivalence classes?), ordering these classes by the number of combinatorial combinations, and solving the cheapest ones first.

For instance, the equivalence class for \( ?e = \text{locatedIn} \) has the following cartesian product between cache and query...
clauses: \{\{?x = locatedIn\}\} \times \{\{?d = locatedIn\}\}, hence
\(?x \rightarrow ?d\) is the only possible mapping.

We can thus simply remove the clauses of the cache from
the query, and insert a reference to the cache: rewrite the
query as: \(?q' = q\)
\(\mu(c) \cup \{C1\}\)

\{\{triple(?a, ?b, ?c),
\{?b = rdf:type\}, \{?c = Bakery\},
\{cache(C1)\}\}

We now seek a mapping from the variables \(\rho\) of the cache to
variables of the query, such the cache cnf becomes a sub-set
of the query cnf.

4.3 Substring matches
Substring matching is a simple method for finding ressources of
interest. By defining an additional subsumed relation
between expressions it becomes possible to use the cache for
this purpose. SPARQL defines a CONTAINS function for
substring matching.

contains(?x , ?cacheVar) subsumedBy contains(?x, ?query-Var) iff CONTAINS(?queryVar, ?cacheVar)

Cache:
{ triple(?x, ?y, ?z), \{?y = rdf:label\}, \{contains(?z, ' hel')\}\}

Query:
{ triple(?s, ?p, ?o), \{?p = rdf:label\}, \{contains(?o, ' h e l l o')\}\}

4.4 Covers
For a given basic filter pattern, multiple cache entries may
be applicable. We abbreviate triple patterns with upper
case letters. For example, consider a query \(Q := A, B, C,\)
and cache entries for \(e1: A, B\) and \(e2: B, C\) and \(e3: C,\)
In fact, as the query is conjunctive, correctness is retained
when applying even overlapping covers, i.e. rewriting \(Q\) as \(Q' := JOIN(e1, JOIN(e2, e3))\). However, from a performance
perspective it creates unnecessary joins.

Finding an optimal set of covers reduces to the Knapsack
problem, which is known to be NP-hard.

4.5 Join order optimization
While formally there is no difference between JOIN(R, S) and
JOIN(S, R), in practice join execution may vary greatly
depending on the sizes of R and S. For instance of R is large
and only few bindings are expected to join with a small
set of bindings S, then it may be better to iterate S and
perform lookups on R. Joins have been greatly studied in the
database field, and there are various ways how they can be
executed in detail. (classic/grace hash join, nested lookup,
...)

5. IMPLEMENTATION

6. IGNORE BELOW

In the following, we restrict our work to the consideration
to the SPARQL operators ordered by their arity:

- DISTINCT
- EXTEND
- PROJECT
- ORDER
- GROUP_BY
- UNION
- FILTER (without the predicates that take graph patterns as arguments - i.e. EXIST)
- LEFT_JOIN
- JOIN Although not needed for joining quad patterns, this is needed for sub queries
- QFPC (this is our own node type that combines quadPattern with filter - quadFilterPatternCanonical)

6.1 View Matching
Second, note that two algebra expressions \(O\) and \(P\) are
equivalent, if there exists a mapping of variables \(\mu: V \rightarrow V\)
such that application of the mapping on \(O\) yields \(P: \mu(O) = P\). This implies that mapping the variables (i.e. the domain)
of the solution bindings of \(\mu([O]) = [P]\)

Assume for a given expression \(P\), the result of \(\mu(P)\) is already
\(\Omega\), and there exists a mapping \(\mu\) such that \(P_\mu = O\),
then it also holds that \([O] = \Omega_{\mu}\), effectively providing the
formal basis for allowing the replacement of expressions with
the result of their evaluation.

7. IMPLEMENTATION

Acknowledgements
GeoKnow