Towards Content-Aware SPARQL Query Caching for Semantic Web Applications

Yanfeng Shu, Michael Compton, Heiko Müller, and Kerry Taylor

Computational Informatics, CSIRO, Australia
{yanfeng.shu,michael.compton,heiko.mueller,kerry.taylor}@csiro.au

Abstract. Applications are increasingly using triple stores as persistence backends, and accessing large amounts of data through SPARQL endpoints. To improve query performance, this paper presents an approach that reuses results of cached queries in a *content-aware* way for answering subsequent queries. With a focus on a common class of conjunctive SPARQL queries with filter conditions, not only does the paper provide an efficient method for testing whether a query can be evaluated on the result of a cached query, but it also shows how to evaluate the query. Experimental results show the effectiveness of the approach.

1 Introduction

With the popularity of Semantic Web technologies, applications are increasingly using triple stores as persistence backends, accessing large amounts of data through SPARQL endpoints. As the number of queries increases, and data grow in size, scalability becomes an issue. To address this, much work has been done to improve the performance of triple stores through better storage, indexing and query optimisation. However, little has been done so far to take advantage of caching.

The work by Martin et al. [6] represents a first step towards filling the gap, where caching is performed by a proxy layer residing between an application and a SPARQL endpoint, and the proxy answers a SPARQL query without accessing the triple store if the query is identical to a cached query. Caching in [6] is basically *content-blind*, unaware of the content of cached results. In this paper, we go one step further and explore reusing cached results in a *content-aware* way, so that the proxy can not only answer a query that exactly matches a cached query, but it can also answer a query by processing the result of a cached query. Such a caching approach requires SPARQL query containment checking, i.e. checking whether the result of a query is contained in that of a cached query. Containment checking for full-SPARQL in general is undecidable [5]. Considering this, we focus on a fragment of SPARQL which is commonly used in real world queries [8], conjunctive queries with simple filter conditions (CQSFs). As our first contribution, we define SPARQL query containment based on the (set) semantics of SPARQL, and give sufficient conditions for containment checking of CQSFs.

Containment checking alone, however, is not enough. It is possible that a query is contained in a cached query but cannot be evaluated on its result. For example,

X. Lin et al. (Eds.): WISE 2013, Part I, LNCS 8180, pp. 320–329, 2013.
© Springer-Verlag Berlin Heidelberg 2013

given query Q_1, returning the names of all students of age 20 from a university database, and query Q_2, returning the names of all students. It is easy to see that the result of Q_1 is contained in that of Q_2. However, Q_1 cannot be evaluated on the result of Q_2, since it does not contain enough information to evaluate the age constraint. Another issue with containment checking is that its cost could be considerable, which potentially compromises the benefit of caching. Our second contribution addresses these two issues. We introduce a notion of *evaluability* and define requirements for queries to be answered using cached results. We further provide an efficient method for checking whether a query can be answered using the result of a cached query (i.e. query evaluability checking), and show how query answering is done. We evaluate our approach experimentally based on the LUBM benchmark. The results show that our approach achieves much better performance than no caching and content-blind caching cases. The rest of the paper is organised as follows. Section 2 introduces the concepts. Section 3 describes our caching approach. Section 4 presents experimental results. Finally, Section 5 concludes the paper and points out future work.

2 Preliminaries

2.1 Syntax and Semantics of SPARQL

SPARQL is the official W3C recommendation for querying RDF graphs. In this paper, we focus on `SELECT` queries on ground RDF graphs. Let V be a set of variables disjoint from U (URIs) and L (Literals). Variables in V are prefixed by the symbol ?. We denote a `SELECT` query by $Q(S, P)$, where $S \subset V$ is the set of variables to be returned, P is the graph pattern to be matched. For simplicity, we restrict our discussion to conjunctive queries with simple filter conditions (CQSFs), i.e. queries composed of `AND` and simple `FILTER` operators. We refer to a filter condition as *simple*, if it involves at most one variable. Given a graph pattern P, we use $vars(P)$ to denote the set of variables in P (for a triple pattern t, we use $vars(t)$), and $ftrs(P)$ to denote the set of filter conditions in P.

We define the semantics of SPARQL by following the set semantics defined in [9,7]. A *solution mapping*[1] μ from V to $U \cup L$ is a partial function $\mu : V \rightarrow U \cup L$. The domain of μ, $dom(\mu)$, is the subset of V where μ is defined. Given a triple pattern t and a solution mapping μ such that $vars(t) \subseteq dom(\mu)$, we use $\mu(t)$ to denote the triple obtained by replacing the variables in t according to u. Two solution mappings μ_1 and μ_2 are compatible, denoted by $\mu_1 \sim \mu_2$, if for all $?X \in dom(\mu_1) \cap dom(\mu_2)$, $\mu_1(?X) = \mu_2(?X)$, i.e. if $\mu_1 \cup \mu_2$ is also a solution mapping. Let Ω, Ω_1 and Ω_2 be sets of solution mappings, R a filter condition, and $S \subset V$ a set of variables. We define algebraic operations join ($\bowtie$), projection (π), and selection (σ) over mapping sets: $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \; and \; \mu_1 \sim \mu_2\}$; $\pi_S(\Omega) = \{\mu_1 \mid \exists \mu_2, \; \mu_1 \cup \mu_2 \in \Omega \wedge dom(\mu_1) \subseteq S \wedge dom(\mu_2) \cap S = \emptyset\}$; $\sigma_R(\Omega) = \{\mu \in \Omega \mid \mu \models R, \; i.e. \; \mu \; satisfies \; R\}$.

[1] It is simply called *mapping* in [9,7].

Based on these operations, the evaluation of graph patterns and queries over an RDF graph G is defined as a function $[[.]]_G$ that takes a pattern or a query, and returns a set of solution mappings. Let G be an RDF graph, t a triple pattern, P, P_1, P_2 graph patterns, R a filter condition, $S \subset V$ a set of variables, and $Q(S, P)$ a `SELECT` query, we define: $[[t]]_G = \{\mu \mid dom(\mu)=vars(t)\ and\ \mu(t) \in G\}$; $[[P_1\ \texttt{AND}\ P_2]]_G = [[P_1]]_G \bowtie [[P_2]]_G$; $[[P\ \texttt{FILTER}\ R]]_G = \sigma_R([[P]]_G)$; $[[Q(S,P)]]_G = \pi_S([[P]]_G)$.

Example 1. Consider a `SELECT` query $Q(\{?x\}, ((?x, type, student)(?x, age, ?y)$ `FILETER` $(?y = 20)))$ and an RDF graph $G = \{(a, type, student), (a, age, 30), (b, type, student), (b, age, 20)\}$. When evaluating Q over G, we obtain $[[Q]]_G = \{\{?x \rightarrow b\}\}$.

2.2 Containment of SPARQL Queries

We define the containment of SPARQL queries based on the definition of subsumption of solution mappings. In [1], the authors introduce a definition of subsumption of solution mappings. Here, we extend their definition by considering different sets of variables that are possibly used in queries.

Definition 1 (Subsumption of Solution Mappings). *Let Ω_1 and Ω_2 be two sets of solution mappings. Ω_1 is subsumed by Ω_2, denoted by $\Omega_1 \sqsubseteq \Omega_2$, if there is a variable mapping ψ from the domain of Ω_1 (the union of domains of its solution mappings) to the domain of Ω_2 that for every $\mu_1 \in \Omega_1$, there exists $\mu_2 \in \Omega_2$ such that $\psi(\mu_1) \subseteq \mu_2$, where $\psi(\mu_1)$ denotes the solution mapping obtained from μ_1 by replacing every variable $?X \in dom(\mu_1)$ with $\psi(?X)$.*

Definition 2 (SPARQL Query Containment). *Let Q and Q' be two SPARQL queries. Q is contained in Q', denoted by $Q \sqsubseteq Q'$, if and only if for every RDF graph G, $[[Q]]_G \sqsubseteq [[Q']]_G$.*

3 Content-aware SPARQL Query Caching

In this section, we describe our caching approach. Similar to [6], the main functionality is performed by a proxy residing between the application(s) and a SPARQL endpoint. Given a query (CQSF), the proxy first parses the query. It then checks whether the query is the same as a cached query by comparing the query strings. If that is the case, the result is retrieved from the cache and returned to the user. If the query is not cached, the proxy checks whether the query can be evaluated on a cached result. Queries that cannot be evaluated on the cache are forwarded to the SPARQL endpoint and results are cached before returned to the user. We use LRU (Least Recently Used) as the replacement scheme for our cache.

Based on results from relational databases [10], we have the following proposition for checking the containment of CQSFs[2].

[2] Without loss of generality, we assume that all the filter conditions are safe, i.e. each variable in a condition appears in some triple pattern.

Proposition 1. *Let $Q(S, P)$ and $Q'(S', P')$ be two CQSFs. $Q \sqsubseteq Q'$ if*

1. *there exists a mapping τ from the variables of P' to the variables, URIs or literals of P that maps each triple pattern of P' to a triple pattern of P,*
2. *$ftrs(P)$ logically implies $\tau(ftrs(P'))$, i.e. $ftrs(P) \Rightarrow \tau(ftrs(P'))$, where $\tau(ftrs(P'))$ denotes the set of filter conditions obtained from $ftrs(P')$ by replacing each variable $?X'$ in $ftrs(P')$ with $\tau(?X')$, and*
3. *$S \subseteq \tau(S')$.*

We refer to a mapping that satisfies Proposition 1 a *containment mapping*. Given two queries Q and Q', $Q \sqsubseteq Q'$ only tells us that the result of Q is contained in that of Q'. We still have to compute the result of Q. Unfortunately, $Q \sqsubseteq Q'$ does not guarantee that Q can be evaluated on the result of Q', as pointed out in the Introduction. Here, we give a definition of what it means that a SPARQL query can be evaluated on the result of another SPARQL query. The definition is inspired by Larson and Yang's work on computing SQL queries from derived relations [4].

Definition 3 (Evaluability). *Let Q and Q' be two SPARQL queries. We say that Q can be evaluated on the result of Q', or simply, Q can be evaluated by Q', denoted by $Q \preceq Q'$, if the operations needed to compute the result of Q from the result of Q' contain no algebraic joins.*

In order for Q to be evaluated on the result of Q', the following conditions are required to hold.

Proposition 2. *Let $Q(S, P)$ and $Q'(S', P')$ be two CQSFs. $Q \preceq Q'$, if*

- *there exists a containment mapping τ from Q' to Q,*
- *for each triple pattern t of P, there exists a triple pattern t' of P' such that $\tau(t') = t$, and*
- *for each variable $?X'$ in P', if $\tau(?X')$ is an URI or a literal, or $\tau(?X')$ is a variable which is included in S or in a filter condition in P, then $?X' \in S'$.*

The first condition is easily understandable: for $Q \preceq Q'$, the result of Q must be contained in that of Q'. The second condition ensures that any triple in an RDF graph that matches a triple pattern of P also matches a triple pattern of P'. The third condition basically specifies the variables that have to be included in S' in order for Q to be evaluated by Q'. These three conditions can be used to terminate testing early if Q cannot be evaluated by Q'.

With this proposition, we can evaluate Q through three types of operations on the result of Q': $\pi_S(\Omega)$, $\sigma_R(\Omega)$, and substitution operation $\tau(\Omega)$. Ω denotes a set of intermediate solution mappings generated during the application of operations, S a set of variables to be returned in Q, and R a conjunction of filter conditions including those in Q, and those derived from τ in the form $?X' = c$ (when a variable in Q' is mapped to an URI or a literal in Q) or $?X' = ?Y'$ (when two different variables in Q' are mapped to the same variable in Q).

Example 2. Consider two queries, $Q_1(\{?X_1\}, (?X_1, type, stu)(?X_1, age, 20))$ and $Q_2\{?X_2, ?Y_2\}, (?X_2, type, stu)(?X_2, age, ?Y_3)FILTER(?Y_2 > 15))$. From $Q2$ to $Q1$, there is a containment mapping τ that satisfies Proposition 2: $\tau(?X_2) = ?X_1, \tau(?Y_2) = 20$. Let G is an RDF graph. We can evaluate Q_1 by Q_2 through $\pi_{\{?X_1\}}(\tau(\sigma_{(?Y_2=20)}([[Q_2]]_G)))$.

To test $Q \preceq Q'$, we need to check whether there is a containment mapping from Q' to Q that satisfies the conditions in Proposition 2. We can do this efficiently if Q' is acyclic, as described below. A key operation is checking whether each triple pattern of Q' can be mapped to a triple pattern of Q. This is done by comparing corresponding elements of triple patterns, i.e. subject to subject, predicate to predicate, and object to object. See Algorithm 1 for details. Let t' be a triple pattern of Q', t a triple pattern of Q, and e' and e a pair of corresponding elements of t' and t. There are four cases: (1) both e' and e are URIs or literals; (2) e' is an URI or a literal, but e is a variable; (3) e' is a variable, but e is an URI or a literal; (4) both e' and e are variables. In the first case, e' and e need to be equivalent in order for t' being able to be mapped to t. In the second case, t' cannot be mapped to t, as e' is more specific than e. In the last two cases, there is a mapping from e' to e. If all the mappings from t' to t are compatible, i.e. they agree on shared variables, then t' can be mapped to t. During the mapping, we can also check whether e' should be included in S', if it is a variable, and whether there are filter conditions involving e' or e, and whether the filter condition involving e' can be implied by the filter condition involving e. In doing so, we are checking whether t could be evaluated by t' with regard to return variables and filter conditions of Q and Q'. In our current implementation, if both filter conditions involving e' and e exist, we require that they be conjunctions of arithmetic comparisons of the form $?X\theta c$, where c is a numeric value, and $\theta \in \{=, \neq, <, \leq, >, \geq\}$.

A triple pattern of Q' can be potentially mapped to several triple patterns of Q. As such, there may be more than one mapping associated with a triple pattern of Q', all with the same domain: the set of the variables in the triple pattern. Such mappings are called *partial mappings* from Q' to Q, as their domains are a subset of the variables of Q'. Based on partial mappings, we are then able to check whether there is a containment mapping from Q' to Q such that $Q \preceq Q'$. Before we present our approach for testing $Q \preceq Q'$, we introduce the concept of query acyclicity.

A query (i.e. CQSF) is *acyclic* (*cyclic*) if its hypergraph is acyclic (cyclic). A query's *hypergraph* consists of a set of vertices and a set of hyperedges: each vertex corresponds to a variable in the query, and each hyperedge corresponds to a triple pattern and includes the variables in the triple pattern. A hypergraph is acyclic if its GYO-reduction results in an empty hypergraph; otherwise it is cyclic. The *GYO-reduction* [2] is a process that repeatedly applies the following two operations on a hypergraph: (1) delete a vertex that occurs in only one hyperedge; (2) delete a hyperedge that is contained in another hyperedge. If a query is acyclic, an elimination tree for the query can be constructed during the GYO-reduction: each node of the tree corresponds to a triple pattern in the

Input: two triple patterns t and t' of Q and Q' respectively,
two sets of return variables S and S' of Q and Q' respectively,
two sets of filter conditions F and F' in Q and Q' respectively
Output: *null*, or a mapping $\tau_{t' \to t}$ from t' to t if $t \preceq t'$

$\tau_{t' \to t} \leftarrow null$, $E \leftarrow$ the triple elements of t, $E' \leftarrow$ triple elements of t';
for $i \leftarrow 1$ ***to*** 3 **do**
  $e \leftarrow$ the ith element of E, $f \leftarrow$ the filter condition involving e in F;
  $e' \leftarrow$ the ith element of E', $f' \leftarrow$ filter condition involving e' in F';
  if *both e and e' are URIs or literals, and $e \neq e'$* **then** *return null*;
  if *e is a variable and e' is not a variable* **then** *return null*;
  if *e is not a variable and e' is a variable* **then**
    if *$e' \notin S'$, or $e' \in S'$, $f' \neq null$ and e does not satisfy f'* **then**
    return null;
  if *both e and e' are variables* **then**
    if $((f = null)$ *and* $(f' = null))$ **then**
      if $(e \in S$ *and* $e' \notin S')$ **then** *return null*;
    if $((f \neq null)$ *and* $(f' = null))$ **then**
      if $(e' \notin S')$ **then** *return null*;
    if $((f = null)$ *and* $(f' \neq null))$ **then** *return null*;
    if $((f \neq null)$ *and* $(f' \neq null))$ **then**
      if *$e' \notin S$, or f' cannot be implied by f after replacing e' in f' with e* **then** *return null*;
  if *e' is a variable* **then**
    if $\{e' \to e\}$ *is compatible with* $\tau_{t' \to t}$ **then** $\tau_{t' \to t} \leftarrow \tau \cup \{e' \to e\}$;
    else *return null*;
return τ;

Algorithm 1. Checking whether $t \preceq t'$

query; if a hyperedge E is deleted by operation (2) because it is contained in some other hyperedge F, then the tree has an edge (t_E, t_F) where t_E denotes the triple pattern corresponding to E, and t_F the triple pattern corresponding to F. If there are several hyperedges containing E when E is deleted, then a random one is picked as F. For simplicity, we restrict our discussion to queries whose hypergraphs are connected. As such, an elimination tree of a query always covers all the triple patterns of the query. However, our results can be generalised to queries with disconnected hypergraphs.

An elimination tree of a query (acyclic) captures relationships of triple patterns in a deterministic way. We can take advantage of this to find containment mappings (if any) efficiently. Since a cyclic query does not have an elimination tree, we have two cases when testing $Q \preceq Q'$, i.e. when Q' is acyclic and when it

Input: two queries Q and Q' (Q' is acyclic)
Output: *null*, or a mapping $\tau_{Q' \to Q}$ from Q' to Q if $Q \preceq Q'$

$T \leftarrow$ *the elimination tree of* Q';
for *each triple pattern* t' *in* T **do**
  $t'.mappings \leftarrow null$;
  for *each triple pattern* t *of* Q **do**
    if $((\tau_{t' \to t} = t \preceq t') \neq null)$ **then** *insert* $\tau_{t' \to t}$ *into* $t'.mappings$;
  return null if no such t*;*
return null if there exists t *of* Q *that cannot be evaluated*;
for *each triple pattern* t' *in* T *(bottom-up)* **do**
  for *each child* t'_i *of* t' **do**
    $t'.mappings \leftarrow t'.mappings \ltimes t'_i.mappings$;
return null if the mappings of T*'s root are empty*;
$\Gamma \leftarrow$ the mappings of T's root;
for *each triple pattern* t' *in* T *(top-down)* **do**
  $\Gamma \leftarrow \Gamma \bowtie t'.mappings$;
  if $dom(\Gamma) = vars(Q')$ **then** **for** *each mapping* $\tau_{Q' \to Q}$ *in* Γ **do**
    if *each triple pattern* t *of* Q *can be mapped to by* $\tau_{Q' \to Q}$ **then**
    return $\tau_{Q' \to Q}$*;*
return null;

Algorithm 2. Checking whether $Q \preceq Q'$

is cyclic (it should become clear in the following that it is unimportant whether Q is acyclic or not). For both cases, the testing consists of two major phases. The first phase is generating partial mappings for each triple pattern of Q', as described earlier. The second phase is generating containment mappings, if any, from partial mappings. If Q' is acyclic, we generate containment mappings by traversing an elimination tree of Q'. We first traverse the tree bottom-up. If a node has children in the tree, we check whether the parent's partial mappings (i.e. the partial mappings associated with the triple pattern corresponding to the parent node) are compatible with its children's by semi-joining the parent's partial mappings with the children's. This is continued until the root of the tree is reached and its partial mappings are processed. If the resulting partial mappings of the root are empty, then there are no containment mappings from Q' to Q such that Q can be evaluated by Q'. Otherwise we traverse the tree top-down to compute containment mappings by joining each node's partial mappings with its children's, until the resulting mappings cover all variables of Q'. Algorithm 2 shows the whole process when Q' is acyclic. It is adapted from the AcyclicContainment algorithm in [2].

Fig. 1. The hypergraph and elimination tree for Q_2 in Example 3

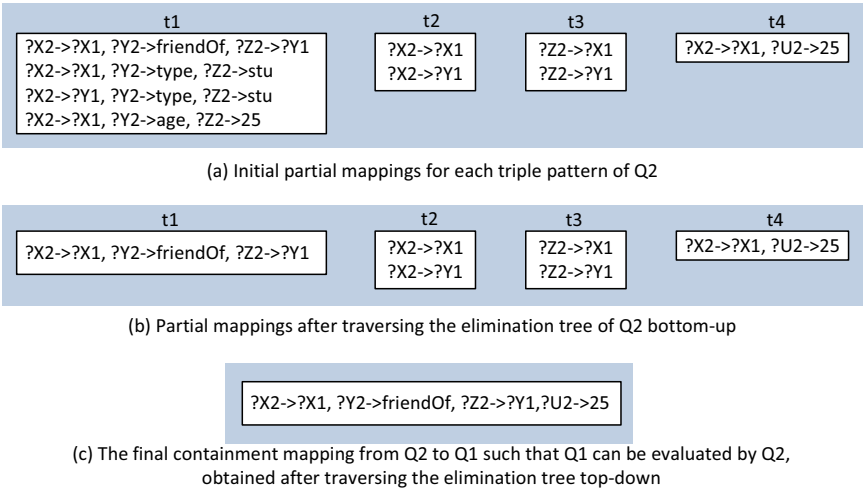

Fig. 2. Mappings generated during testing $Q_1 \preceq Q_2$ in Example 3

If Q' is cyclic, we need to check each triple pattern's partial mappings against all other triple patterns' if their domains overlap, and compute the partial mappings through semi-join operations. This is continued until there is a triple pattern whose partial mappings are empty, or there are no more changes to all triple patterns' partial mappings. If it is the former case, it means that there are no containment mappings from Q' to Q such that Q can be evaluated by Q'. In the latter case, we compute containment mappings by joining all triple patterns' partial mappings with overlapping domains one by one until the resulting mappings cover all variables of Q'. Though some heuristics can be applied to decide which triple patterns' partial mappings are processed first, the testing of $Q \preceq Q'$ when Q' is cyclic is in general inefficient in both time and space. Fortunately, most real world queries are acyclic and have few triple patterns [8].

Example 3. Let Q_1 be $(\{?X_1, ?Y_1\}, (?X_1, friendOf, ?Y_1)(?X_1, type, stu)(?Y_1, type, stu)(?Y_1, age, 25))$, Q_2 be $(\{?X_2, ?Y_2, ?Z_2\}, (?X_2, ?Y_2, ?Z_2)(?X_2, type, stu)(?Z_2, type, stu)(?X_2, age, ?U_2)FILTER(?U_2 > 20))$. To test $Q_1 \preceq Q_2$, we first

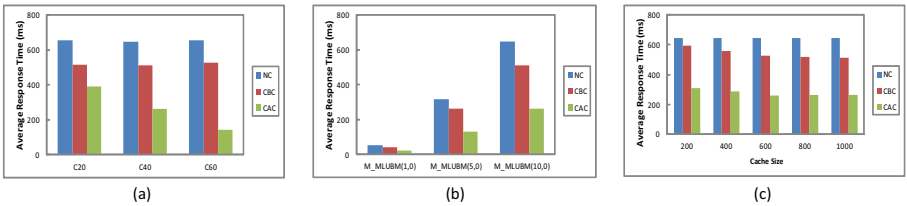

Fig. 3. Performance comparison of the three cases with respect to percentage of contained queries, dataset size, cache size

construct the hypergraph and an elimination tree for Q_2, as shown in Figure 1, where t_1, t_2, t_3, and t_4 represent triple patterns $(?X_2, ?Y_2, ?Z_2)$, $(?X_2, type, stu)$, $(?Z_2, type, stu)$, $(?X_2, age, ?U_2)$ respectively. We then generate the containment mappings from Q_2 to Q_1 such that $Q_1 \preceq Q_2$. The generation process is shown in Figure 2(a)-(c). Let τ be the final mapping in (c), G be an RDF Graph. We can evaluate Q_1 on the result of Q_2 through $\pi_{\{?X_1,?Y_1\}}(\tau(\sigma_{(?U_2=25)(?Y_2=friendOf)}([[Q_2]]_G)))$.

4 Evaluation

We evaluated the performance of our approach through experiments. All experiments were done on a machine with the following configuration: Intel Core i5 (M540, 2.53GHz), 3.24 GB of RAM, 848GB HD, Java 1.6, 512MB of max heap size, Apache Jena 2.7.3, and TDB 0.9.3 (with the default file caching).

We generated datasets by modifying the LUBM benchmark [3]. We added a data property "value" (with the range of "xsd:double") to the benchmark ontology (univ-bench.owl) and changed the data generator so that each generated student instance would have a certain random "value". Using the modified data generator, we generated 3 datasets: MLUBM(1,0), MLUBM(5,0) and MLUBM(10,0), which contain OWL files for 1, 5, and 10 universities respectively, and then loaded each dataset into TDB after OWL inferencing. The sizes of their materialised versions are 185182, 1125339, 2279105 respectively. For query generation, we used the following template: SELECT ?X ?Y WHERE{?X rdf:type ub:Student. ?X ub:value ?Y FILTER (?Y >= %%value1%% && ?Y<= %%value2%%) ?X ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0>}. By controlling the values which are used to replace the parameters, we generated query traces with different percentages of contained queries (i.e. the queries that are contained in other queries in the same trace). The traces we experimented with are C20, C40, and C60, with 20%, 40%, and 60% of contained queries respectively. Each trace contains 1,000 queries and has the same percentage of identical queries (20%).

Figure 3(a) shows the average response time of the three traces on M_MLUBM (10,0) in three cases, i.e. no caching (NC), content-blind caching (CBC), and content-aware caching (CAC). As expected, when the percentage of contained queries increases, the average response times with CAC decrease. This, however,

has little impact on the performance of CBC, as the percentage of identical queries is the same for the three traces. We then studied the performance of CAC with respect to dataset size. For this, we ran C40 to the three datasets. As shown in Figure 3(b), CAC outperforms CBC by 44%-50%, and the performance improvement is even better for larger datasets. In the earlier experiments, we assumed unlimited cache size, i.e. there is no cache replacement. To investigate the behavior of CAC with respect to cache size, we ran C40 on M_MLUBM(10,0) at various cache sizes. From Figure 3(c), we see that even with a small cache size, CAC achieves much better performance than CBC. Also, CAC seems to be more resilient to the change of cache size.

5 Conclusions and Future Work

In this paper, we presented a caching approach for improving the performance of Semantic Web applications. Our approach is novel in that not only does it benefit a query that exactly matches a cached query, but it also benefits a query that is contained in a cached query and can be evaluated by the cached query. We tested our approach on the slightly modified version of the LUBM benchmark. Experimental results showed that our approach can achieve much better performance than no caching and content-blind caching cases. In the future, we would like to try some other cache replacement schemes, e.g. those considering access frequency or miss cost. Also, we plan to extend our approach for other fragments of SPARQL, e.g. queries with `OPT` and `UNION` operators.

References

1. Arenas, M., Perez, J.: Querying Semantic Web Data with SPARQL. In: Proceedings of PODS (2011)
2. Chekuri, C., Rajaranman, A.: Conjunctive Query Containment Revisited. Theoretical Computer Science 239(2), 211–229 (2000)
3. Guo, Y., Pan, Z., Heflin, J.: LUBM: a Benchmark for OWL Knowledge Base Systems. Journal of Web Semantics 3(2), 158–182 (2005)
4. Larson, P.A., Yang, H.Z.: Computing Queries from Derived Relations. In: Proceedings of VLDB (1985)
5. Letelier, A., Perez, J., Pichler, R., Skritek, S.: Static Analysis and Optimisation of Semantic Web Queries. In: Proceedings of PODS (2012)
6. Martin, M., Unbehauen, J., Auer, S.: Improving the Performance of Semantic Web Applications with SPARQL Query Caching. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010, Part II. LNCS, vol. 6089, pp. 304–318. Springer, Heidelberg (2010)
7. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 30–43. Springer, Heidelberg (2006)
8. Picalausa, F., Vansummeren, S.: What are Real SPARQL Queries Like? In: Proceedings of SWIM (2011)
9. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL Query Optimization. In: Proceedings of ICDT (2010)
10. Ullman, J.D.: Principles of Database Systems, 2nd edn. Computer Science Press (1982)