Improving the Performance of Semantic Web Applications with SPARQL Query Caching

Michael Martin, Jörg Unbehauen, and Sören Auer

Universität Leipzig, Institut für Informatik, Augustusplatz 10-11,
D-04109 Leipzig, Germany
`lastname@informatik.uni-leipzig.de`
`http://aksw.org`

Abstract. The performance of triple stores is one of the major obstacles for the deployment of semantic technologies in many usage scenarios. In particular, Semantic Web applications, which use triple stores as persistence backends, trade performance for the advantage of flexibility with regard to information structuring. In order to get closer to the performance of relational database-backed Web applications, we developed an approach for improving the performance of triple stores by caching query results and even complete application objects. The selective invalidation of cache objects, following updates of the underlying knowledge bases, is based on analysing the graph patterns of cached SPARQL queries in order to obtain information about what kind of updates will change the query result. We evaluated our approach by extending the BSBM triple store benchmark with an update dimension as well as in typical Semantic Web application scenarios.

1 Introduction

It has been widely acknowledged that the querying performance of triple stores is a decisive factor for the large-scale deployment of semantic technologies in many usage scenarios (cf. e.g. [9,4]). In recent years much progress has been made to improve the performance of triple stores by developing better storage, indexing and query optimization. However, compared to querying data stored in a fixed relational database schema, querying a triple store is still usually slower by a factor of 2-20 (cf. e.g. BSBM results[1]). This shortcoming is due to the fact that columns in a relational database are typed and may be indexed more efficiently. By using a triple store, this efficiency is lost to the flexibility of amending and reorganizing schema structures easily and quickly.

A circumstance currently not yet taken advantage of by triple stores is that in typical application scenarios only relatively small parts of a knowledge base change within a short period of time. The majority of triples remain unchanged. Hence, most queries will return the same results even after the occurrence of changes on the knowledge base. In addition, queries are often frequently issued,

[1] `http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/`

for example, when different users access the same information in a Semantic Web application. We can take advantage of this fact by caching query results, but also want to ensure that cached query results are selectively invalidated on knowledge base updates.

An analysis of a query shows what kind of changes of the knowledge base it will take to return a different result. In the meantime the results of the query can be temporarily stored for fast access. Our approach is based on examining SPARQL graph patterns. A query result is cached as long as updated triples do not match any of the triple patterns comprised by the graph pattern. Once an updated triple matches any of the triple patterns, the corresponding cache object is invalidated and will have to be recomputed by the triple store on a subsequent execution of the query.

Web applications are often composed out of smaller objects whose state depends on the execution of multiple queries. The product description page of an online shop, for example, is composed of header and footer components, a product category selection menu, the actual product description and possibly personal information of the actual user, such as the contents of his/her shopping cart etc. Traditional Web applications cache application objects or even whole parts of the generated user interface (i.e. HTML page fragments). The application logic then has to take care of invalidating these complex cache objects, for example, when new products are entered into the system or the user's shopping cart changes. We allow the caching and invalidation of more such compound application objects by associating them with all of the cached query results they depend on. The compound cache object is then invalidated when any of the associated query results change.

As a result, Semantic Web applications which frequently issue the same queries and are updated moderately are significantly accelerated. This improvement allows Semantic Web applications to get closer to conventional Web applications based on relational databases with regard to performance. In particular, we make the following contributions:

- We provide a method for selective invalidation of cached query results on triple store updates based on an analysis of SPARQL queries.
- We extended the caching of plain query results into a caching of compound application objects, based on a dependency tacking.
- We implemented the RDF query caching approach as a small proxy layer which resides between the Semantic Web application and an arbitrary SPARQL/SPARUL endpoint.
- We extended the BSBM triple store benchmark to consider updates and evaluated our approach with both the synthetic benchmark as well as in a practical Semantic Web application setting.

The paper is structured as follows: We describe the concepts and architecture of our caching solution in the Sections 2 and 3, while elaborating on the cache maintenance in Section 4. We also provide a comprehensive evaluation of the approach based on a synthetic benchmark as well as a real Semantic Web applications in Section 5. We conclude and present related as well as future work in the Sections 6 and 7.

2 Concepts

In this section we describe the theoretical foundation of our approach. It is based on the SPARQL algebra and we refer to [8,7] for a more detailed description of the algebraic formalization of SPARQL. We will briefly introduce the formal SPARQL syntax and semantics and then derive a proposition about the invariance of graph pattern solutions when updates of the underlying RDF dataset do not match any of the triple patterns used in the graph pattern.

2.1 Syntax and Semantics of SPARQL

The SPARQL query language is based on the definition of the syntactic language features and a semantic interpretation of these syntactic features by means of set theoretical operators. We restrict ourselves to the core fragment of SPARQL over simple RDF (i.e. RDF without RDFS vocabulary and literal rules), which is sufficient for our purposes.

Syntax. Assume there are pairwise disjoint infinite sets I, B, and L (IRIs, Blank nodes, and RDF literals, respectively). A triple $(v_1, v_2, v_3) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an RDF *triple*. In this tuple, v_1 is the *subject*, v_2 the *predicate* and v_3 the *object*. We denote the union $I \cup B \cup L$ as by T called RDF terms. Additionally, we assume the existence of an infinite set V of variables which is disjoint from the above sets. An RDF *graph* is a set of RDF triples (also called RDF dataset, or simply a dataset).

A SPARQL graph pattern expression is defined recursively as follows:

1. A tuple from $(T \cup V) \times (I \cup V) \times (T \cup V)$ is a graph pattern (a *triple pattern*).
2. If P_1 and P_2 are graph patterns, then expressions (P_1 AND P_2), (P_1 OPT P_2), and (P_1 UNION P_2) are graph patterns.
3. If P is a graph pattern and R is a SPARQL condition, then the expression (P FILTER R) is a graph pattern.

SPARQL conditions are supposed to evaluate to boolean values. Additionally, we assume that for (P FILTER R) the condition $var(R) \subseteq var(P)$ holds, where $var(R)$ and $var(P)$ are the sets of variables occurring in R and P respectively.

Semantics. A mapping μ from V to T is a partial function $\mu : V \rightarrow T$. For a triple pattern t we denote as by $\mu(t)$ the triple obtained by replacing the variables in t according to μ. The domain of μ, $dom(\mu)$, is the subset of V where μ is defined. Two mappings μ_1 and μ_2 are *compatible* when for all $x \in dom(\mu_1) \cap dom(\mu_2)$ we have $\mu_1(x) = \mu_2(x)$, i.e. when $\mu_1 \cup \mu_2$ is also a mapping. Note that two mappings with disjoint domains are always compatible and that the empty mapping (i.e. the mapping with empty domain) $\mu_\emptyset$ is compatible with any other mapping. Let Ω_1 and Ω_2 be sets of mappings. We define the join of, the union of and the difference between Ω_1 and Ω_2 as:

1. $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ are compatible mappings$\}$,
2. $\Omega_1 \cup \Omega_2 = \{\mu | \mu \in \Omega_1$ or $\mu \in \Omega_2\}$,
3. $\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 |$ for all $\mu_2 \in \Omega_2$, μ_1 and μ_2 are not compatible$\}$.

Now we can define the semantics of graph pattern expressions by means of a function $[[\cdot]]_D$, which takes a pattern expression and returns a set of mappings. As in [7], we assume, for the reason of simplicity, all datasets to be free of redundancies (i.e. duplicate triples).

Definition 1 (Graph pattern evaluation). *Let D be an RDF dataset over T, t a triple pattern, R a SPARQL condition and P_1, P_2 graph patterns. Then the evaluation of a graph pattern over D, denoted as by $[[\cdot]]_D$, is defined recursively as follows:*

1. $[[t]]_D = \{\mu | dom(\mu) = var(t)$ *and* $\mu(t) \in D\}$,
2. $[[(P_1\ AND\ P_2)]]_D = [[P_1]]_D \bowtie [[P_2]]_D$
3. $[[(P_1\ OPT\ P_2)]]_D = ([[P_1]]_D \bowtie [[P_2]]_D) \cup ([[P_1]]_D \setminus [[P_2]]_D)$
4. $[[(P_1\ UNION\ P_2)]]_D = [[P_1]]_D \cup [[P_2]]_D$
5. $[[(P_1\ FILTER\ R)]]_D = \{\mu \in [[P_1]]_D | R(\mu)$ *evaluates to boolean true* $\}$

Note that we omitted a detailed description of the semantics of filter expressions for the purpose of brevity here. The elements μ of the result of an evaluation are also called solutions of the respective graph pattern.

2.2 Graph Pattern Solution Invariance

After we defined the syntax and semantics of SPARQL, we now investigate under which types of updates the results of SPARQL graph patterns change. This analysis lays the theoretical foundation for our query result caching framework, since a certain query result can be cached until an update of the underlying RDF would affect this particular query result. Speaking intuitively the solution of a graph pattern stays the same at least until a triple, which matches any of the triple patterns being part of the graph pattern, is added to or deleted from the RDF dataset.

Proposition 1 (Graph pattern solution invariance). *If Ω is the set of all solutions for the graph pattern P with respect to a dataset D and for a triple t there exists no mapping μ from query variables to RDF terms such that $t \in \mu(P)$, then Ω is also the set of all solutions for $D_+ = D \cup \{t\}$ and $D_- = D \setminus \{t\}$.*

Proof. We first show (a) that the proposition holds when P is a triple pattern and then (b) that the evaluation of a graph pattern does not change if the sets of all solutions for the triple patterns contained in the graph pattern do not change.

(a) We assume, P is a triple pattern and there is a solution μ of P with regard to D_+. According to the graph pattern evaluation (1) holds $\mu(P) \in D_+$. According to our precondition $t \notin \mu(P)$. Consequently, μ is a solution for $D_+ \setminus \{t\} = D$ and hence $\mu \in \Omega$. The proof for D_- proceeds accordingly.

(b) The evaluation of graph patterns consisting of AND, OPT and UNION clauses (i.e. points 2-4 in Definition 1) is defined to be composed out of the solutions of the constituting graph patterns via the join, union and difference operators. Hence, if the set of solutions for D equals the sets of solutions for D_+ (D_-) for the constituting graph patterns, so will the set of solutions for their composition. A similar argument holds for the application of a filter clause (i.e. point 5 in Definition 1): If $[[P_1]]_D = [[P_1]]_{D_+}$ ($[[P_1]]_D = [[P_1]]_{D_-}$), then the set of solutions for the filter clause stay the same, i.e. $[[(P_1 \text{ FILTER } R)]]_D = [[(P_1 \text{ FILTER } R)]]_{D_+}$ ($[[(P_1 \text{ FILTER } R)]]_D = [[(P_1 \text{ FILTER } R)]]_{D_-}$). $\square$

3 Architecture

In order to employ the invariance of graph pattern solutions for caching, we have to be aware of all queries as well as of all dataset updates. Hence, we implemented our approach as a small proxy layer, which resides between the Semantic Web application and the SPARQL/SPARUL [10] endpoint. All SPARQL queries and SPARUL updates are routed through this proxy. Once the proxy receives a query, it checks whether a result for this query is cached in its local store. If that is the case, the result is directly delivered to the client without accessing the triple store. If the query was not previously stored and is not excluded from caching by user-supplied rules, the query is routed to the triple store and, before results are returned to the client, these are stored in the cache's local result store.

We developed two implementations of the SPARQL cache: Firstly, we integrated the cache as a component into the Erfurt API[2] - a middleware for Semantic Web applications used as foundation for OntoWiki [5]. As a second implementation, we developed a stand alone version in Java, which can be used in conjunction with arbitrary Semantic Web applications and SPARQL endpoints. Both implementations are evaluated and compared in Section 5.

4 Cache Population and Maintenance

Other than conventional Web application caching approaches, we have to accomodate two requirements: (1) we want to invalidate cache objects not only based on a unique identifier or predefined timespans, but selectively on updates of the triple store. (2) In addition to simple query results, we want to store more complex application objects which are composed out of the results of multiple queries.

In oder to accomodate these requirements, the cache object store is implemented based on a relational database. The ER diagram is visualized in Figure 1. Query results are stored in a serialized form in the `cache_query_result` table. Optionally, they are associated to surrounding cache objects stored in the `cache_object` table. Query results are firstly associated with RDF models the query is accessing and secondly, for fast invalidation, the triple patterns comprised by the graph patterns of the query are stored in the table

[2] `http://aksw.org/Projects/Erfurt`

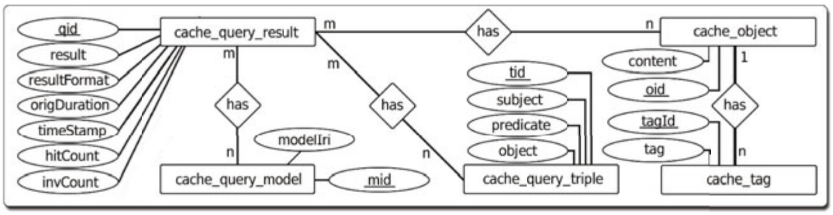

Fig. 1. ER diagram of the cache's relational object store

`cache_query_triple`. Variables in the triple patterns are represented as `NULL` values in this table.

4.1 Storing and Loading of Query Results

The general operation of our SPARQL cache is visualized in Figure 2. Once our SPARQL caching proxy receives a query, it computes the queries MD5 hash to determine quickly whether the query has already been cached or has to be (re-)executed. If the query result has not yet been stored in the cache, the SPARQL query is parsed and handed over to the original SPARQL endpoint. The returned result is stored together with the parsed query adhering to the cache schema. Currently, we use a relational database and in-memory backends to store cache objects.

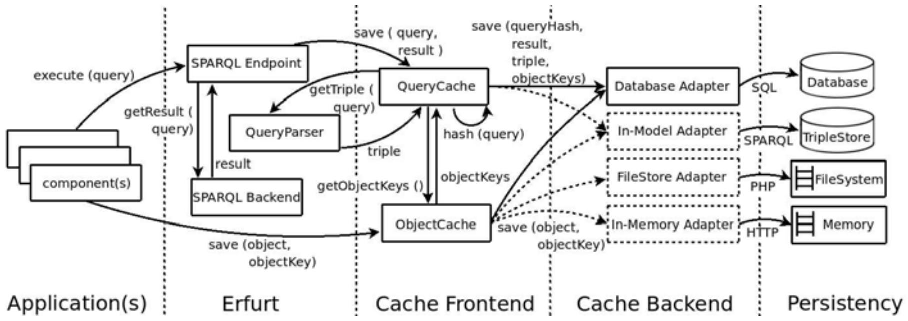

Fig. 2. Querying a SPARQL endpoint and storing the result using the query cache

Listing 1.1 shows an example query containing three triple patterns. Table 1 shows the rows which are added to the `cache_query_rt` and `cache_query_triple` tables. Since multiple queries might contain the same triple patterns, we store triple patterns only once and associate them with the queries (cf. Figure 1).

```
PREFIX  aksw: <http://aksw.org/people#>
PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT  ?classUri ?classLabel FROM <http://aksw.org/people#>
WHERE { ?classUri  rdfs:subClassOf  aksw:People .
        ?classUri  aksw:sort  ?sort .
        OPTIONAL { ?classUri  rdfs:label  ?classLabel } }
```

Listing 1.1. Example SPARQL query

Table 1. Extracted triple pattern from SPARQL query

cache_query_rt	
qid	**tid**
1	1
1	2
1	3

cache_query_triple			
tid	**subject**	**predicate**	**object**
1	NULL	rdfs:subClassOf	aksw:People
2	NULL	aksw:sort	NULL
3	NULL	rdfs:label	NULL

4.2 Storing and Loading of Application Objects

In addition to caching query results, our cache implementation offers to cache arbitrary application objects. For this, the cache implementation offers two functions `cacheStart($key)` and `cacheStop($key,$cacheObject)`. When the first function is called, the cache checks whether a cache object for `$key` exists and, if existent, returns this object. If a cache object is not available for the respective `$key`, an entry in the cache object store is created (table `cache_object` in Figure 1) and all subsequent SPARQL queries are associated with this cache object until the function `cacheStop` is called and the respective cache object content is stored.

4.3 Cache Maintenance

The graph pattern solution invariance as derived in Section 2 allows us to invalidate cache objects on triple store updates selectively, assuming that all updates (i.e. insertions or deletions of triples) are routed through the SPARQL query cache proxy. When a certain triple is inserted (or deleted) according to Proposition 1, we have to invalidate all SPARQL queries which contain a triple pattern matching the inserted (or deleted) triple. In addition, we invalidate all compound cache objects, which depend on one or more of the invalidated SPARQL query results.

Please note that the invalidation removes stored query results, but keeps the stored query structure and statistics (e.g. hit_count, inv_count) intact so that a subsequent execution of the query can reuse this information.

We illustrate the process with the addition of the following triple:

```
(G,S,P,O) = (http://aksw.org/people#,aksw:Student,rdfs:subClassOf,aksw:Person)
```

The following SQL query is subsequently used to invalidate query results:

```
UPDATE cache_query_result SET result=NULL WHERE ( qid IN (
SELECT DISTINCT(qid) FROM cache_query_rt JOIN cache_query_triple
                     ON cache_query_rt.tid = cache_query_triple.tid
WHERE ( (
( subject   = 'aksw:student'    OR subject   IS NULL ) AND
( predicate = 'rdfs:subClassOf' OR predicate IS NULL ) AND
( object    = 'aksw:Person'     OR object    IS NULL )
) ) ) AND qid IN (
 SELECT DISTINCT (qid) FROM cache_query_rm JOIN cache_query_model
                       ON cache_query_rm.mid = cache_query_model.mid
 WHERE ( cache_query_model.modelIri = 'http://aksw.org/people#' OR
         cache_query_model.modelIri IS NULL ) ) )
```

5 Evaluation

We measured the impact of our caching solution on the querying performance in two scenarios. First we employed the Berlin SPARQL Benchmark (BSBM, [4]) to demonstrate the cache's abilities by using a well-known test procedure. The second evaluation scenario measures the performance improvements for the Semantic Web application Vakantieland.

All benchmarking was done on a machine with the following configuration: Intel Core 2 Duo (P8400: 2x2.276GHz), 2x2GB of RAM, 160GB SATA HD (7,200rpm), Ubuntu 9.04 32 bit, Java 1.6, PHP 5.2.10, OpenLink Virtuoso 5.09 (NumberOfBuffers=300000, MaxDirtyBuffers=50000).

5.1 Berlin SPARQL Benchmark

The Berlin SPARQL Benchmark (BSBM) is based on an e-commerce use case, simulating an end-user search for products, vendors and reviews. The resulting SPARQL queries are grouped into mixes, each one consisting of 25 queries. The queries are derived from twelve different types and are instantiated by replacing parameters with concrete, randomized values. The QueryMixes per Hour (QMpH) assessment then states how many of these query mixes a certain triples store is able to execute per hour.

While in the original benchmark the probability for selecting a specific parameter is equal for each parameter, we chose to have the parameters selected according to the Pareto distribution, since this reflects practical use cases better and enables us to measure the performance gain of our caching solution in such scenarios. The probability density function can be described by (cf. [15]):

$$P(x) = \frac{ab^a}{x^{a+1}}$$

The parameter a defines the shape, whereas b defines the minimum value. Applied to the benchmark scenario, this implies that we have a number of products or offers that are queried more often than others. In our benchmark adoption we varied the parameter a in order to see how well the caching implementation

Table 2. Distribution of the parameter in dependency to the choice of a

Distr. parameter	**linear**	**a=0.1**	**a=0.3**	**a=0.5**	**a=1.0**	**a=2.0**	**a=4.0**
Unique queries	11718	6205	4147	2953	1694	624	142
Res. distribution	50.5/49.5	64/36	72/28	78/22	84/16	88/12	90/10

adopts to a wider or narrower spectrum of repeated queries. For the Pareto principle (commonly known also as the 80/20 rule of thumb), Table 2 shows how the choice of a broadens the distribution of the parameter (based on a benchmark with 10 million triples and 12,500 queries).

For example, using a linear distribution for creating 12,500 requests results in the generation of 11,718 unique queries, which represents a very limited number of repeated queries (i.e. 782). As for the other extreme of $a = 4$ a total of only 142 unique queries is generated and 90% percent of the 12,500 requests are executed with just the 10% of the unique queries, resulting in a very high level of repetition. Querying was parallelized by using 5 threads totaling 500 query mixes, which is similar to the original BSBM benchmark. For our benchmark adoption we used the Stochastic Simulation in Java library [3] to generate random numbers adhering to the Pareto distribution.

While the Java implementation of the cache was benchmarked as a SPARQL endpoint we also wanted to determine the performance impact on Web applications. Hence, for the Erfurt implementation time was measured by using JMeter[4]. The query duration was then transformed into QueryMixes per Hour (QMpH) for better comparability.

Impact of Distribution. This scenario demonstrates the impact of the query distribution on the performance. We tested various dataset sizes, ranging from 1 million to 25 million with various distributions. The results are summarized in Figure 3.

As expected, we found that the distribution of the queries has a high impact on performance. Performance can benefit enormously when using the cache with a high level of query repetition. For $a > 2$ performance is nearly independent of the store size, since most of the queries can be answered from the cache. Applications with a lower number of repeating queries may not benefit as much, in the Erfurt implementation, for broader distributions with $a \leq 0.3$ the SPARQL cache is not (yet) able to improve performance. For scenarios with larger datasets (≥ 10 million triples) and a moderate query repetition ($1.0 \geq a \geq 0.3$), performance improvements between 12% and 151% are possible.

The second parameter evaluated here is the store size, which, together with the distribution, defines the point at which an application benefits from using a cache. For the Erfurt implementation using a store with 1 million triples, caching offers improved performance starting at $a = 1$, whereas a larger store with 25 million triples profits much earlier, i.e. already from $a = 0.3$ onwards.

[3] `http://www.iro.umontreal.ca/~simardr/ssj/`

[4] `http://jakarta.apache.org/jmeter/`

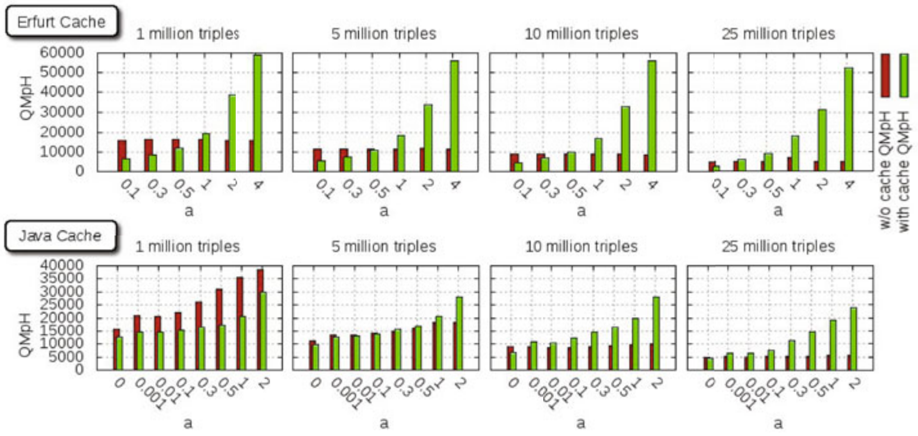

Fig. 3. Impact of distribution parameter on query performance

Most notably regarding the Java implementation is that, when accessed directly via SPARQL, Virtuoso shows similar performance gains with increasing query repetition in the 1 million triples scenario. With greater store size this effect diminishes. For larger store sizes, however, even for relatively low values of a performance gains can be noted, which can be attributed to the light overhead the cache imposes on queries. With no query repetition the cache generates an overhead of just 8% up to 25% in the worst case.

Impact of Updates. Since cache objects will become stale, when the underlying dataset changes, we extended BSBM to support the modification of the queried graph. We removed a number of triples from the original graph in order to load them later, during querying, into the graph. This was implemented by adding inserts into some query mixes, with each insert containing 5 to 8 statements. Thus, we can show how the invalidation of cache objects and the subsequently required re-issuing of the query affects the performance. We compared the impact of different update frequencies with and without caching for the 5 (Erfurt) and 10 (Java) million triples dataset and with the query distribution parameter $a = 1$ (Erfurt) and $a = 0.3$ (Java). The update frequency is determined by the rate of query mixes containing an insert statement.

The results, as depicted in Figure 4, first contain a reference value without inserts, where the cache enabled version executes 60% more QMpH. For the Erfurt cache this advantage is slightly affected by an insert included in every 100th query mix, reducing the performance gain to 48%. With an insert included in every 10th query mix, the performance gain drops to 33%. Including an insert in every query mix and thus every 25th query being an insert statement, reduces performance by 9% compared to the direct use of Virtuoso. For the Java implementation we measured slightly different results. The effect on performance due to the update queries is here stronger, the performance gain drops to 17%, 6%

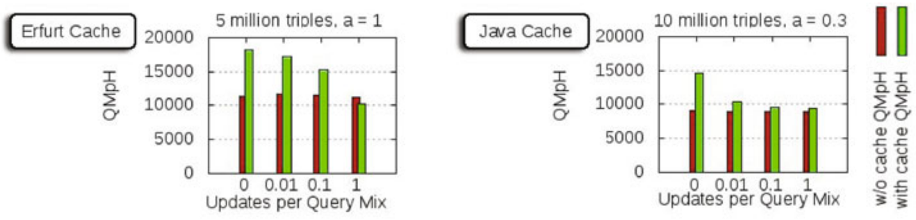

Fig. 4. Impact of update frequency on query performance

and 5%, but does not fall below the values of the same queries without cache. We attribute this behavior to the different cache integration approach.

While the figures presented here show that update frequency and the resulting cache maintenance affect the performance, most Web applications update frequencies are rather at the lower end of the tested values and will thus still significantly benefit from caching. Likewise, with larger datasets the positive impact of caching will be even more noticeable.

5.2 Benchmarking the Semantic Web Application Vakantieland

We evaluated the performance of the SPARQL query and application object caching with the Semantic Web application *Vakantieland*[5]. Vakantieland publishes comprehensive information about 20,000 touristic points-of-interest (POI) in the Netherlands such as textual descriptions, location information and opening hours. The information is stored in a knowledge base containing almost 2 million triples and is structured using approximately 1,250 properties as well as 400 classes. Vakantieland was designed according to the model/view/controller principle and uses the Erfurt API as middleware. Almost all of the information presented in Vakantieland is retrieved using SPARQL. Figure 5 marks areas of the Vakantieland user interface, which are significantly facilitated by the cache. Area 1 and 2 contain different category trees. The first is modelled hierarchically using `owl:class` and `rdfs:subClassOf`. The second category tree represents administrative areas of the Netherlands, containing provinces, districts and cities. For rendering both hierarchies, recursively executed SPARQL queries are used. The remaining two areas (numbered 3 and 4 in Figure 5) provide a collection of POIs (3) and a pagination for navigating over them (4). POIs are presented depending on given filter criteria. These criteria can be free text search, a class or a spatial-area selection, a map-bounding box or a combination of these. Every POI description, which can also be visited on a separate details page, consists of properties arranged in a property hierarchy using `rdfs:subPropertyOf`. These property hierarchies are also obtained using a set of recursively executed SPARQL queries, whose performance was substantially improved by the cache. For automatically benchmarking the behaviour of Vakantieland in combination

[5] Currently available at: `http://staging.vakantieland.nl`

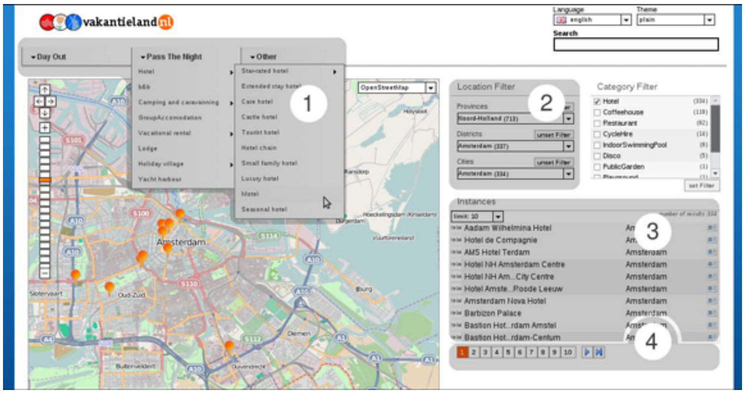

Fig. 5. Vakantieland GUI with marked UI components

with the cache, we defined the following usage scenarios. In every usage scenario we simulated the usage of the pagination, as depicted in Figure 5 (area 4), 200 times with different offsets.

– **Usage Scenario A** No selected filter criteria; no updates.
– **Usage Scenario B** Selected spatial-area filter; updating the `rdfs:label` of the selected spatial filter after every 25th request),
– **Usage Scenario C** Selected class filter; adding a new point-of-interest after every 25th request (`rdf:type` relation of the new resource is identical to the selected category filter),
– **Usage Scenario D** Selected tourism category filter; updating the `rdfs:label` of the selected category after every 25th request),

The results of these use cases are presented in Table 3 and show that the cache proxy implementation improves performance substantially. The application is accelerated between factor 5 (in scenario C) and factor 13 (in scenario A). For improving the performance of SPARQL queries, which filter information by geo-coordinates or search terms, we generated the additional index P, G, S, O on the table RDF_QUAD in Virtuoso[6]. Due to the use of this index, such SPARQL queries can be executed five times faster. Other tested indexes do not measurably improve request times.

6 Related Work

To the best of our knowledge only few approaches exist aiming at improving query performance of RDF stores by means of query caching. We consequently also examine the use of caching in relational, DB-based Web applications and examine how they relate to our solution.

[6] Described at `http://docs.openlinksw.com/virtuoso/rdfperformancetuning.html`

Table 3. Benchmark results of application Vakantieland

Benchmark		**Sceanrio A**	**Sceanrio B**	**Sceanrio C**	**Sceanrio D**
Cache disabled	**pt**	≈ 5480ms	≈ 5262ms	≈ 3967ms	≈ 3992ms
	c	⊘ 428	⊘ 297	⊘ 344	⊘ 344
Warm-Up	**pt**	≈ 3561ms	≈ 3432ms	≈ 3043ms	≈ 3042ms
with enabled **QC**	**qch**	⊘ 358	⊘ 249	⊘ 286	⊘ 286
	qci		≈ 0,32	≈ 1,16	≈ 0,24
Hot Run	**pt**	≈ 2048ms	≈ 2550ms	≈ 2152ms	≈ 1901ms
with enabled **QC**	**qch**	⊘ 420	⊘ 289	⊘ 335	⊘ 336
	qci		≈ 0,32	≈ 1,12	≈ 0,24
Warm-Up	**pt**	≈ 4152ms	≈ 2783ms	≈ 3019ms	≈ 3088ms
with enabled **QC** and **OC**	**qch**	⊘ 162	⊘ 48	⊘ 91	⊘ 91
	qci		≈ 0,32	≈ 1,11	≈ 0,24
Hot Run	**pt**	≈ 403ms	≈ 477ms	≈ 686ms	≈ 434ms
with enabled **QC** and **OC**	**qch**	⊘ 6	⊘ 8	⊘ 5	⊘ 6
	qci		≈ 0,32	≈ 1,12	≈ 0,24

QC:***Q****uery* ***C****ache*; **OC**:***O****bject* ***C****ache*;
pt:***p****rocess* ***t****ime per request*; **c**:*query* ***c****ount per request*;
qch:***q****uery* ***c****ache* ***h****its per request*; **qci**:***q****uery* ***c****ache* ***i****nvalidation per request*

Caching in Web Applications. Caching, which, in contrast to database replication, relies on intercepting queries, is distinguished in [13] into two different approaches: Content-Aware Caching (CAC) and Content-Blind Caching (CBC).

Content-Aware Caching (CAC) systems create upon intercepted queries new Partially Materialized Views (PMVs). This approach is, for example, implemented by DBproxy [3] or MTCache [6]. Whenever a query is executed, the CAC system checks whether the query is entailed by previously cached content and in case it is, the result is computed upon that or the query is forwarded to the database server. By proposing a query federation system, DBCache [2] is further able to relay non-cached parts of a query to the main database.

Content-Blind Caching (CBC) systems, in contrast, are not aware of the structure of the cached data. As demonstrated in GlobeCBC [12], storing only the result and meta-information can be an efficient approach, especially in scenarios with a high query repetition, as costs associated with the subsumption checks can be avoided. Our caching approach can, therefore, be considered to be a CBC system, as the query results are opaque to the cache and are not modified. The systems introduced here rely for cache maintenance and invalidation on database replication mechanisms which notify the caches on updates. While our invalidation mechanism is triggered by intercepted SPARUL queries, integrating this mechamism into data base replication is a possiblity for future work.

Caching for Semantic Web Applications. In [16] a write-through cache holding triples with commonly used subjects is described. Furthermore, property tables as a storage scheme for RDF is introduced, similar to the idea of a vertical partitioning of an RDF store [1]. Used for frequently reoccurring query patterns, this concept can be transferred into creating an RDF Content-Aware Cache. With an intelligent materialization algorithm, the proliferation of tables, as discussed in [11], should be avoidable.

For caching in client-server or peer-to-peer and distributed database environments, query containment algorithms are developed to reuse a query result by subsuming a distinct query. In [14] this approach is applied to RDF stores. It is based on the notion of similarity of RDF queries determined by the costs of transform the results of a previous query into the result for the actual one. The paper discusses the problem of subsumption for RDF queries, presents a cost model and derives a similarity measure for RDF queries based on the cost model and the notion of graph edit distance. The author further sees the strong need to develop strategies for building and maintaining the cache, i.e. changes in the stored information have to invalidate parts of the cached results, as is the main contribution of our approach.

7 Conclusions and Future Work

We presented a novel approach for caching the results of querying triple stores and compound application objects containing such queries. The approach is based on the observation that large parts of a knowledge base usually do not change over time and hence only a small part of the query results are affected by updates to the knowledge base. By identifying the affected query results we are able to selectively invalidate cache objects on updates of the knowledge base such that the cache never contains outdated cache objects. We were able to show that our approach outperforms cacheless triple stores in realistic usage scenarios by more than factor 10. Only in scenarios with small knowledge bases (<1M triple) or very infrequent query repetition our cache adds some overhead. Currently our implementation is only loosely coupled with the underlying triple store. By tighter integrating the cache with the triple store even higher performance gains will be possible.

Future Work. We consider this work as an initial step towards closing the performance gap between relational database and triple store based applications. In order to further exploit the possibilities of caching we aim at looking how the results of a cached query can be reused in a content aware way for answering subsequent queries. In particular, the evaluation of SPARQL filter conditions is a promising candidate for further speed improvements.

Most triple stores are meanwhile equipped with support for light-weight inferencing. While our caching strategy will work well with forward-chaining reasoning approaches (the inferencing of new triples can be simply considered as updates) it still remains to explore how it can be combined with backward chaining inferencing. Another promising direction of future work in the context of the emerging Linked Data Web is how our caching approach can be employed for the acceleration of distributed and federated queries.

References

1. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable semantic web data management using vertical partitioning. In: VLDB. ACM, New York (2007)
2. Altinel, M., Bornhvd, C., Krishnamurthy, S., Mohan, C., Pirahesh, H., Reinwald, B.: Cache tables: Paving the way for an adaptive database cache. In: VLDB (2003)
3. Amiri, K., Park, S., Tewari, R., Padmanabhan, S.: Dbproxy: A dynamic data cache for web applications. In: ICDE (2003)
4. Bizer, C., Schultz, A.: The berlin SPARQL benchmark. International Journal on Semantic Web and Information Systems (2009)
5. Heino, N., Dietzold, S., Martin, M., Auer, S.: Developing semantic web applications with the ontowiki framework. In: Networked Knowledge - Networked Media. SCI, vol. 221. Springer, Heidelberg (2009)
6. Larson, P.-Å., Goldstein, J., Guo, H., Zhou, J.: Mtcache: Mid-tier database caching for SQL server. IEEE Data Eng. Bull. 27(2) (2004)
7. Perez, 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. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Recommendation (2008), `http://www.w3.org/TR/rdf-sparql-query`
9. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: Sp2bench: A SPARQL performance benchmark. In: ICDE. IEEE, Los Alamitos (2009)
10. Seaborne, A., Manjunath, G.: SPARQL/Update - a language for updating RDF graphs (2008), `http://www.w3.org/Submission/SPARQL-Update/`
11. Sidirourgos, L., Goncalves, R., Kersten, M.L., Nes, N., Manegold, S.: Column-store support for RDF data management: not all swans are white. PVLDB 1(2) (2008)
12. Sivasubramanian, S., Pierre, G., van Steen, M., Alonso, G.: GlobeCBC: Content-blind result caching for dynamic web applications. Technical Report IR-CS-022, Vrije Universiteit, Amsterdam, The Netherlands (June 2006)
13. Sivasubramanian, S., Pierre, G., van Steen, M., Alonso, G.: Analysis of caching and replication strategies for web applications. IEEE Internet Computing 11, 60–66 (2007)
14. Stuckenschmidt, H.: Similarity-based query caching. In: Christiansen, H., Hacid, M.-S., Andreasen, T., Larsen, H.L. (eds.) FQAS 2004. LNCS (LNAI), vol. 3055, pp. 295–306. Springer, Heidelberg (2004)
15. Weisstein, E.W.: Pareto distribution. MathWorld - A Wolfram Web Resource (2009)
16. Wilkinson, K., Sayers, C., Kuno, H.A., Reynolds, D.: Efficient RDF storage and retrieval in Jena2. In: SWDB, pp. 131–150 (2003)