Enabling fine-grained HTTP caching of SPARQL query results

Gregory Todd Williams and Jesse Weaver

Tetherless World Constellation, Rensselaer Polytechnic Institute, Troy, NY, USA
{willig4,weavej3}@cs.rpi.edu

Abstract. As SPARQL endpoints are increasingly used to serve linked data, their ability to scale becomes crucial. Although much work has been done to improve query evaluation, little has been done to take advantage of caching. Effective solutions for caching query results can improve scalability by reducing latency, network IO, and CPU overhead. We show that simple augmentation of the database indexes found in common SPARQL implementations can directly lead to effective caching at the HTTP protocol level. Using tests from the Berlin SPARQL benchmark, we evaluate the potential of such caching to improve overall efficiency of SPARQL query evaluation.

1 Introduction

SPARQL endpoints are increasingly being used to provide access to large amounts of linked data. As use increases, both in frequency and complexity, and as the amount of data being served increases, scaling these systems to handle the increased load is crucial. Much work has been done on improving performance of SPARQL processors through more intelligent query planners, optimized index structures, and parallelization. However, there has been little work on addressing scalability through the use of caching.

Caching of query results can benefit both the SPARQL endpoint and the client. When a client uses a conditional HTTP request to which the server responds with a "Not Modified" message, only the IO for the response header is required. On the server, validating a conditional request is likely to be faster and require fewer resources (both CPU time and working memory) than evaluating the whole query. This allows the server to respond to more and/or more complex queries given fixed resources (or, conversely, response to the same queries with fewer resources). Moreover, if successfully validating a conditional request is faster than evaluating the query, the client benefits not only from reduced IO but also reduced latency and potentially by avoiding the need to parse the response (if a client's local cache is able to store a parsed representation).

The benefits of caching are only realized if both the client and server support the caching protocol and if requests are repeatedly made for already-cached results. Client-side support for caching is already available due to the widespread support in HTTP libraries that are used to implement the SPARQL protocol.

In the rest of this work, we propose how to enable support for caching in the data structures used on the server. Because of the widespread support for HTTP caching, and the high frequency of repeated queries, caching of SPARQL query results has the potential to significantly improve efficiency. We restrict our work to only consider caching at the HTTP level as the standard SPARQL Protocol is defined in terms of HTTP[1].

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 reviews the caching features of HTTP. Section 4 defines "relevant" data as it pertains to caching the result of a SPARQL query pattern, and introduces the data structures and algorithms necessary for enabling caching in SPARQL query evaluation. Section 5 presents experimental results showing the effects of caching using the Berlin SPARQL Benchmark (BSBM). Section 6 concludes and discusses possible future work.

2 Related Work

Using caching to increase scalability touches upon several areas research, including statistical distributions of queries affecting their cacheability, indexing structures used in Semantic Web query answering systems, and caching as it relates to both the Semantic Web and to databases. In this section we discuss work related to these areas.

Regarding the repetition of queries, work on analyzing web access logs by Breslau, et al.[1] found that the statistical distribution of requests followed a "Zipf-like distribution" with the distribution exponent varying between different user communities. This finding suggests that caching can have a significant impact on real-world access patterns because a large portion of requests are made for a small set of resources. More recently, and related specifically to SPARQL requests, Gallego, et al.[2] analyzed a set of SPARQL endpoint query logs, and found a high degree of queries duplicated from the same hosts. However, Gallego, et al. only mention the repeated queries in passing, without specific details on the distribution of repeated queries.

There has been a trend in SPARQL systems to use search trees to efficiently index RDF data (following similar use in relational databases) and to use many indexes to support a range of access patterns. The YARS system[3] (and subsequently Hexastore[4] and RDF-3X[5]) demonstrated the effectiveness of maintaining many search tree indexes to provide direct access to RDF data matching a certain triple- or quad-pattern. YARS made use of six B+ tree indexes ($\langle SPOG\rangle$, $\langle POG\rangle$, $\langle OGS\rangle$, $\langle GSP\rangle$, $\langle GP\rangle$, $\langle OS\rangle$) to cover all sixteen possible quad-access patterns. Hexastore and RDF-3X, while only considering triples, both made use of six indexes ($\langle SPO\rangle$, $\langle SOP\rangle$, $\langle PSO\rangle$, $\langle POS\rangle$, $\langle OSP\rangle$, $\langle OPS\rangle$) to provide complete indexing of triples, covering all eight triple access patterns

[1] The SPARQL 1.0 Protocol is defined in terms of WSDL with bindings for HTTP and SOAP. However, the SPARQL 1.1 Protocol is defined only for HTTP as there was no widespread support for non-HTTP implementations.

and all possible orderings. The design of 4store[6] makes use of only three indexes ($\langle PS \rangle$, $\langle PO \rangle$, $\langle G \rangle$), using RADIX tries for the $\langle PS \rangle$ and $\langle PO \rangle$ indexes[2]. While all of these systems utilize search trees for performance, their use is restricted only to indexing the RDF data. In our work, we make use of the search trees not only to maintain many indexes over the RDF data but also to store additional metadata about when that data was modified. As described in the following sections, keeping such metadata allows a query processor to validate existing cached query results.

Caching database query results has been studied widely. Goldstein and Larson[7] show the potential of materializing views within a relational system to dramatically improve performance of expensive queries. Both Amiri, et al.[8] and Larson, Goldstein, and Zhou[9] addresses caching relational query results using materialized views. Amiri, et al. perform caching in edge caches separate from the origin database which reduces load on the origin server, but maintaining consistency of cached results requires that the origin database propagate every update, delete, and insert operation to all caches, making it unsuitable for environments with high write throughput. Larson, Goldstein, and Zhou improve upon the approach by Amiri, et al. by allowing more flexible materialization of views, allowing the query optimizer to choose whether to evaluate the query on the origin server even in the presence of cached data, and improving support for parameterized queries. In contrast to these approaches, the fixed structure of RDF data makes supporting caching much easier. No complex logic or knowledge of database schemas is needed to determine which tables or columns might benefit from caching as all data is structured in terms of triples.

Caching as it relates to the Semantic Web is a much more recent field of study. The work on caching SPARQL query results by Martin, Unbehauen, and Auer[10] shares the same goal and many details with our work (we frequently cite this work herin as a source of common groundwork and greater detail) However, they perform caching by coupling a caching layer with an existing SPARQL processor. This has the benefit of portability across SPARQL implementations, but incurs high cache maintenance costs and is suitable only for caches which are tightly integrated with the underlying system and can intercept all write operations (e.g. ISP caches or caches built into a user agent cannot be used). Finally, the work in [10] deals with only a subset of SPARQL; in this work we make specific note of several features of SPARQL that deserve special attention in the context of caching.

Work by Hartig[11] shows performance improvements of using a local cache in evaluating queries by linked data link traversal. This work is complementary to ours in that both use caching to improve performance of query answering, one over static linked data, the other over potentially dynamic data available via structured queries.

---

[2] 4store actually maintains two RADIX tries for each predicate, but for our purposes these may be understood as being equivalent to tries with P prepended to the actual keys.

3 HTTP Caching

In this section we introduce the caching features available in HTTP upon which our system relies.

HTTP supports two primary caching mechanisms, allowing servers to explicitly indicate a caching expiration (with an `Expires` date or a `max-age` duration) or indicating a cache validator (with a `Last-Modified` date or `ETag` value). Here we concern ourselves only with cache validators – specifically, `Last-Modified` dates – as they are a more natural fit for caching data that may be updated in the database at any time. However, as they relate to our work, both the `Last-Modified` and `ETag` headers may be understood as being effectively equivalent as we do not use the more expressive "weak validation" that ETags allow.

The `Last-Modified` validator works as follows. A client user-agent requests a resource (in this case a SPARQL query) to the server:

```
GET /sparql?query=SELECT... HTTP/1.1
Host: example.org
```

The server sends back a response whose header includes the `Last-Modified` validator with a date indicating when the resource was last modified:

```
HTTP/1.1 200 OK
Last-Modified: Wed, 1 Jun 2011 12:45:15 GMT
Content-Type: application/sparql-results+xml

<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
...
</sparql>
```

At some point in the future, the client requests the resource again and, noting that the response is cached from the last time it was requested, indicates the request as *conditional* by using the `If-Modified-Since` header with the previously returned validator date:

```
GET /sparql?query=SELECT... HTTP/1.1
Host: example.org
If-Modified-Since: Wed, 1 Jun 2011 12:45:15 GMT
```

If the resource has not changed since the validator date, the server sends a response indicating that the already-cached content is still valid:

```
HTTP/1.1 304 Not Modified
```

Otherwise, the server responds as usual with a full response, including the resource content and any applicable cache validators (the updated `Last-Modified` time).

In terms of SPARQL, HTTP caching ought to make query results appear as valid ("fresh") so long as the query results do not change. Since determining precisely if results to a query have changed may require re-evaluating the query (negating one of the benefits of caching), we settle for a less strict condition: caching ought to make the query results appear as valid so long as data "relevant" to the query have not changed. Once a query result has been cached, updates to "irrelevant" data in the SPARQL endpoint should have no affect on the caching – upon re-submitting the query, the server should indicate that the cached results are still valid. "Relevant" data being updated prior to the query being re-submitted should result in the server re-evaluating the query and returning fresh query results. In section 4.2 we define "relevant" and "irrelevant" data.

4 Methodology

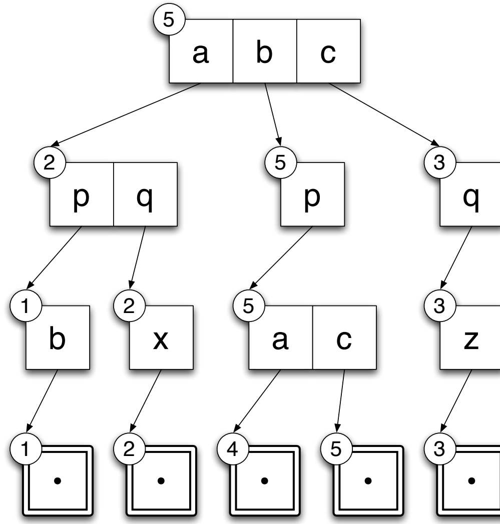
We propose modifying the search trees used to index RDF data in a simple way to enable determining the effective modification time of data relevant to a query. In the following sections we show how the modification time data stored in the search trees can be maintained during database updates, and how the data can be retrieved and used at query time. In determining what data is relevant to a specific query, we extend the work done by Martin, Unbehauen, and Auer[10] (what they call "Graph Pattern Solution Invariance") to support the much more expressive queries and graph patterns of SPARQL 1.1[3]. This includes the use of named graphs, property paths, and `DESCRIBE` queries.

In this work we assume that the SPARQL processor is built using a quad-store, and that the SPARQL "RDF Dataset" is mapped directly to the statements in the quad-store (with a special graph name representing the default graph). This assumption simplifies the following discussion, but is not required by our approach. The algorithms we present can be extended to work with arbitrary mappings between RDF dataset and quad-store, or to work with graph-stores.

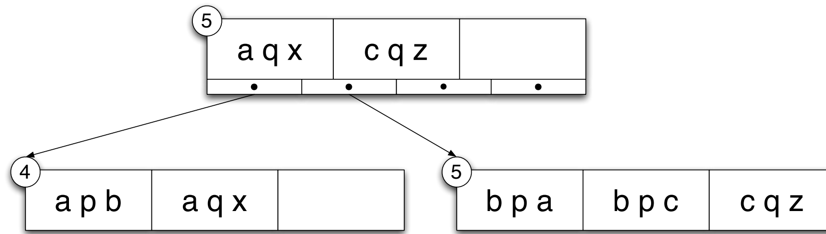
4.1 Search Tree Indexing

Search trees are a common data structure used to implement efficient access to RDF data for varying access patterns. To determine the modification-time (mtime) of "relevant" data in a search tree, we propose adding an mtime field to each node in the search tree. During an update operation (insertion or deletion of RDF data), we update the mtime field in each affected search tree node. Moreover, during an update we ensure the mtime of each node in the tree is greater than or equal to the mtime of all of its children. In this way, the mtime of a node in the tree can be used as a conservative proxy value for the mtime of any of its descendant nodes.

[3] http://www.w3.org/TR/sparql11-query/



(a) Trie Index



(b) B+ Tree Index

Fig. 1: Example $\langle SPO \rangle$ search trees

For each access pattern in a query, we can now determine an mtime after which we can be assured that no update operation has affected data matching the pattern. By calculating the maximum mtime over all the query access patterns, we arrive at a single timestamp which is at least as recent as the actual modification time of the query's "relevant" data.

While different SPARQL systems make use of different types of search trees, and use varying numbers of indexes, we propose a general solution that works with any number of indexes and across a variety of tree types (we discuss specifics of both B+ trees and tries). Although the caching results in our proposed system are complete, soundness is affected by the choice of a specific search tree type and number of available indexes. For example, fewer indexes, or the use of B+ trees versus tries, may cause some query results to appear as if they have changed

when in fact they haven't. However, query results that are asserted as being the same as cached results will always in fact be the same.

Figure 1 shows both a B+ tree of order 4 and a trie with example data[4]. The mtime of each tree node appears inside the circle attached to each node, and shows the mtimes that result from this example 5-update sequence (with mtimes starting at 1, and incrementing on subsequent operations):

1. Insert triple { `a p b` }
2. Insert triple { `a q x` }
3. Insert triple { `c q z` }
4. Insert triple { `b p a` }
5. Insert triple { `b p c` }

As can be seen, the trie maintains the correct mtime for each leaf node while the mtimes in the B+ tree only indicate that the triples with subject `a` were not updated by operation 5. We discuss the reasons for this unsoundness in section 4.3.

4.2 Relevant data and graph patterns

We define data "relevant" to a SPARQL query as being data that *may* affect the results of the query. Martin, Unbehauen, and Auer claim: "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 [graph]." This is true when considering only *triple*-patterns in the default graph, but to support the full expressivity of SPARQL, we must extend this claim: The solution of a query with 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 default graph
- a triple, which matches any of the triple patterns being part of a `GRAPH <iri>` pattern, is added to or deleted from the `<iri>` named graph
- a triple, which matches any of the triple patterns being part of a `GRAPH ?var` pattern, is added to or deleted from any named graph
- a triple is added to a new named graph and the graph pattern includes an empty `GRAPH ?var` pattern
- a triple is removed from a named graph, leaving the graph empty, and the graph pattern includes an empty `GRAPH ?var` pattern
- a triple, with predicate matching any part of a property path being part of the graph pattern, is added to or deleted from the dataset
- a triple is added or removed from the dataset, and the graph pattern includes a zero-length or negated property path
- a triple is added to or deleted from the dataset, and the query uses the `DESCRIBE` form

We discuss each of these cases and how they relate to relevant data below.

[4] The example data used here is comprised of triples for brevity; the handling of mtimes is identical for quad data.

Named Graph Patterns As noted in [10], the addition or deletion of a triple (to the default graph) may change the solutions of a graph pattern. To fully support SPARQL datasets (which contain not just the default graph, but also any number of named graphs), we must also consider graph patterns scoped to a named graph. These patterns may either be scoped to a specific named graph (using the GRAPH <iri> { ... }) syntax) or be scoped to *any* named graph (using the GRAPH ?var { ... } syntax). For a graph pattern scoped to a specific named graph, iri, the solutions to the graph pattern may change with the addition or deletion of a triple matching the graph pattern to the named graph iri. For a pattern scoped to any named graph, the solutions to the pattern may change with the addition or deletion of a triple matching the graph pattern to any named graph.

Empty Named Graph Patterns Beyond graph patterns scoped to named graphs, special handling is required for the *empty* named graph pattern:

```
SELECT ?g WHERE { GRAPH ?g {} }
```

This query returns the set of graph names in the dataset. The query has no triple-patterns which might match triples being added or removed, yet its results may change based on added or removed data. Specifically, adding a triple to a new named graph, or removing the final triple from a named graph[5] may change the solutions to this pattern.

Paths Property paths greatly increase the expressiveness of SPARQL, but as they relate to relevant data, may be reduced to the matching of triple patterns. We can partition property paths into two categories: fixed-length and variable-length. Fixed-length property paths are those that can be syntactically represented by basic graph patterns (BGPs). Due to their equivalence with BGPs (sets of triples), these paths can be handled the same as triple patterns.

Variable-length paths are those that cannot be reduced to BGPs, and may rely on new algebraic operations to match data. These include zero-length paths (?s <p>{0} ?o), zero-or-more paths (?s <p>* ?o), one-or-more paths (?s <p>+ ?o), and negated paths (?s !<p> ?o). Due to their complexity, we discuss only a subset of the expressivity of these path types.

With respect to "relevant" data, one-or-more paths with simple predicates (those in which the + path operator applies to an IRI) can be reduced to triple pattern matching with predicate-bound triple patterns. For example, the path <s> <p>+ ?o has the same relevant data as the triple pattern ?s <p> ?o. Note that while the subject is bound to <s> in the path pattern, it must be unbound in the triple pattern equivalent as the path may be affected by triples where the subject is not <s>.

[5] Some SPARQL systems allow empty named graphs to exist. Removing all triples from a named graph would not affect the set of graph names on such systems.

Zero-length paths and negated paths require special attention. The zero-length path connects a graph node (any subject or object in the graph) to itself. Therefore, any insertion (deletion) in a graph may affect the results to a zero-length path pattern by adding (removing) a node to the graph that didn't exist before (doesn't exist after) the update. Similarly, a negated path `?s !<p> ?o` implies that any insertion or deletion *not* using the `<p>` predicate may impact the results. While the relevant data for such a negated path is a subset of all the data, we assume that realistic datasets will contain a range of predicates and so the relevant data will be very large (in many cases approximating the size of the dataset itself). Therefore, we conservatively assume that the entire dataset is relevant to a negated path pattern.

DESCRIBE Queries `DESCRIBE` queries present a challenge in determining relevant data. These queries involve matching a graph pattern just as `SELECT` queries do (a `DESCRIBE` query without a `WHERE` clause being semantically equivalent to one with an empty `WHERE` clause). However, the final results of a `DESCRIBE` query depend on the `WHERE` clause *and* the algorithm used for enumerating the RDF triples that comprise the description of a resource.

A naïve `DESCRIBE` algorithm would be to return all the triples in which the resource appeared as the subject. For our purposes, this algorithm would make this query:

```
DESCRIBE ?s
WHERE { ?s a <Class> }
```

roughly equivalent to a `SELECT` query with an additional triple pattern:

```
SELECT ?s ?p ?o
WHERE { ?s a <Class> .
        ?s ?p ?o }
```

Since most `DESCRIBE` algorithms will include *at least* these triples, and given the course-grained nature of the triple pattern `?s ?p ?o` (matching every triple in the database), we consider the "relevant" data for a `DESCRIBE` query to be all data in the database. We note that the work in [10] does not (and need not) address this issue as that work is concerned with caching of *graph pattern* results, not *query* results. Since the `DESCRIBE` query form takes *graph pattern* results (or ground IRIs) as input, and outputs an implementation-dependent set *query* results, the caching of *query* results must respect this process.

Given that the algorithm used for `DESCRIBE` queries is implementation dependent, our definition of "relevant" data for `DESCRIBE` queries is intentionally conservative and we do not discuss specific handling of `DESCRIBE` queries in any further detail.

4.3 Maintaining and Probing Cache Status

In this section we briefly describe the algorithms used during update operations to maintain the mtime field in the search tree. We then describe the probing algorithm used to determine the effective mtime of the relevant data for a specific query.

Cache Maintenance Maintaining the mtime field in the search tree is a simple process:

1. Before each tree node is written to disk (due to an insertion of deletion), update the node's mtime to the current time.
2. For each node that is written to disk, write its parent to disk (thereby updating its parent's mtime).

This process will ensure our condition that every tree node's mtime is greater or equal to those of its descendants and can be used as the effective mtime of descendant, relevant data.

We distinguish between the effective mtime of data matching an access pattern, and that data's actual mtime. As discussed in section 4.1, the specific data structure used for the search tree affects the granularity (and therefore the expected accuracy) of the effective mtime. Due to their design, tries yield effective mtimes that are exactly the same as the most recent mtime of data matching an access pattern. B+ trees yield effective mtimes of matching data that may be affected by any non-matching data that is co-located on a leaf node with matching data.

During the update process, we note that the parent node(s) may already need to be written to disk (in the case of a node split), so step 2 may already be required on any given update. Moreover, an update at a leaf node in append-only and counted B+ trees cause a cascade of writes up to the (possibly new) root. In these cases, all IO incurred by the cache update algorithm is already required by the update operation, and so the cost of maintaining the cache data is effectively free.

Cache Probing The algorithm used for probing a database index to retrieve the effective mtime for a query is shown in algorithm 1. Given a query and a set of available search tree indexes, for each access pattern in the query, the algorithm probes the index that will yield the most accurate effective mtime, and returns the most recent of the mtimes. The index that will yield the most accurate effective mtime is the one with a key ordering that will allow descending as deep into the tree as there are bound terms in the access pattern. If no such index exists, a suitable replacement index is chosen that maximizes the possible depth into the tree that some subset of bound terms in the access pattern will allow. In the case of the completely unbound access pattern, the effective mtime is the same as the mtime of the entire dataset and so can be retrieved from the root node of *any* available index.

While this algorithm describes how the effective mtime of a query may be computed, it is worth noting that the specific steps described may be implemented in more or less efficient ways. For example, the algorithm calls for finding the lowest common ancestor (LCA) of data matching the access pattern. For a system using B+ trees, a naïve implementation might traverse the tree to find the leaves with matching data and then walk up the tree to find the LCA. A more efficient implementation could avoid having to find all leaves with matching data by traversing tree edges until finding the LCA by using the bounds data contained in internal nodes.

Algorithm 1: Probe database for effective mtime of query results

Input: A SPARQL query graph pattern *query*, a set of available database indexes *indexes*
Output: $effectiveMtime$, the effective modification time of relevant data for the query

```
1  mtimes = ∅
2  foreach ap ∈ query do
3      orderedIndexes = {i|i∈indexes, ∃s⊆boundPositions(ap) s.t. the key order
       of i starts with s}
4      if |orderedIndexes| > 0 then
5          index = argmax_{i∈orderedIndexes} |s|
6          n = LCA of data matching ap in index
7          mtimes = mtimes ∪ {mtime(n)}
8      else
9          i = any index in indexes
10         mtimes = mtimes ∪ {mtime(root(i))}
11     end
12 end
13 effectiveMtime = Max(mtimes)
14 return effectiveMtime
```

As discussed above and in section 4.1, the soundness of results is affected by the choice of the search tree data structure used. B+ trees produce less sound results as a result of maintaining less accurate effective mtimes. Tries will result in more sound results as a result of being able to maintain accurate effective mtimes. Even though tries maintain accurate effective mtimes, their use does not guarantee perfectly sound cache validation as updates that affect data relevant to a query may not change the results to that query. This can occur when the relevant updated data does not appear in the query results due to join conditions, filter expressions, or projection. In these cases, cache validation will fail and the query must be evaluated again, despite accurate results already being cached.

One final case that is worth noting is the special case of determining the effective mtime for an *empty* named graph pattern (GRAPH ?g). As discussed

in section 4.2, this pattern returns the set of available named graphs. If the set of available indexes are all covering indexes (using key orders that are just permutations of subject, predicate, object, and graph), then there is no way to determine an accurate effective mtime for this pattern. However, if there is an available index over just $\langle G \rangle$, an accurate effective mtime for the set of named graphs is stored in the $\langle G \rangle$ index root node.

5 Evaluation

We evaluated the potential impact on performance of query result caching by implementing a simple SPARQL process in C with B+ tree indexes. To index data, we use the six index orderings $\langle SPOG \rangle$, $\langle SGOP \rangle$, $\langle POGS \rangle$, $\langle OGSP \rangle$, $\langle OSPG \rangle$, and $\langle GPSO \rangle$. We evaluated our system using a slightly modified version of the Berlin SPARQL Benchmark[6] using both the Explore (read-only) and the Explore and Update (read-write) use cases. All BSBM evaluation was performed on a dual Intel Xeon E5504 Quad Core 2.0GHz processor with 24GB of memory, with 5 warmup runs, and 10 timed runs.

5.1 Modified Berlin SPARQL Benchmark

We believe the standard BSBM benchmark fails to account for the skewed distribution of real-world queries and so, following the work in Martin[10], modified the benchmark test driver to use a Pareto distribution for benchmark queries. The evaluation tests were performed with varying query repetition as represented by the α parameter. We also modified the benchmark test driver to support HTTP caching by storing query results when they are returned with caching headers, and validating existing cached query results using conditional requests.

5.2 Explore Use Case

The Explore use case of BSBM consists of a set ("query mix") of read-only queries that simulate a consumer looking for product information in an e-commerce setting. In our evaluation, this use case tests performance gains from caching on a static dataset. No updates (neither relevant nor irrelevant) are performed and so once cached, query results are always valid.

Figure 2 shows the performance improvement of our caching system on the BSBM Explore use case (as a percentage increase over the same tests run without the use of caching). The test was run with α distribution values ranging from 0.1 to 4.0, and shows between 35–650% increase in benchmark performance.

5.3 Explore and Update Use Case

The Explore and Update use case of BSBM consists of the same queries as in the Explore use case, with occasional updates to the dataset representing new

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

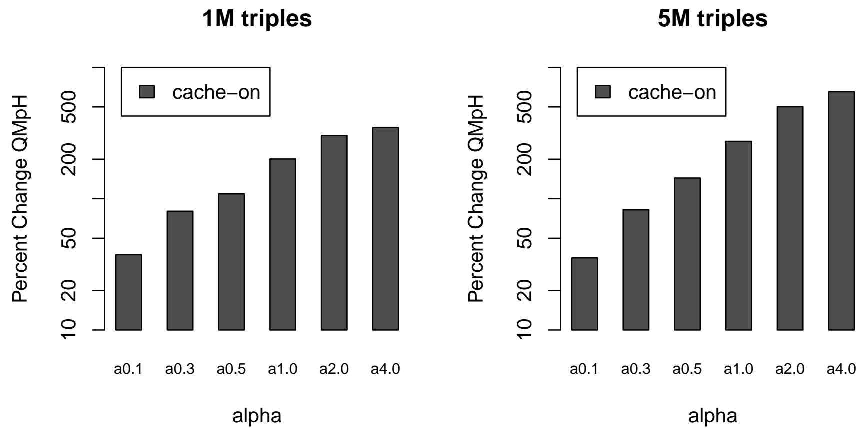


Fig. 2: BSBM Explore Use Case

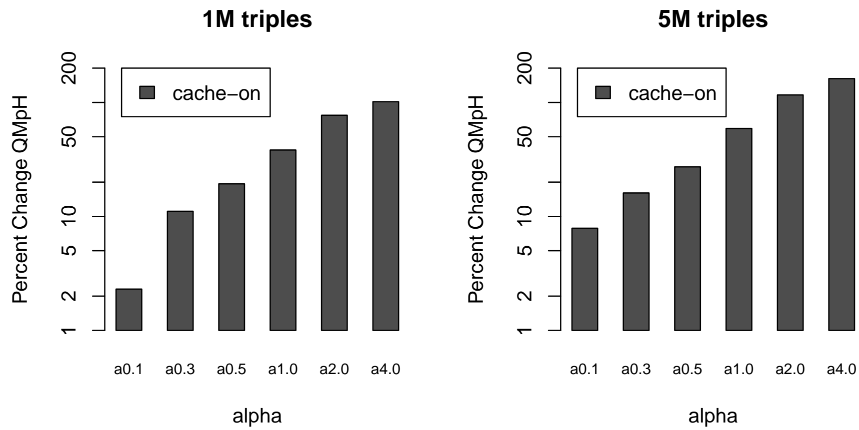


Fig. 3: BSBM Explore and Update Use Case

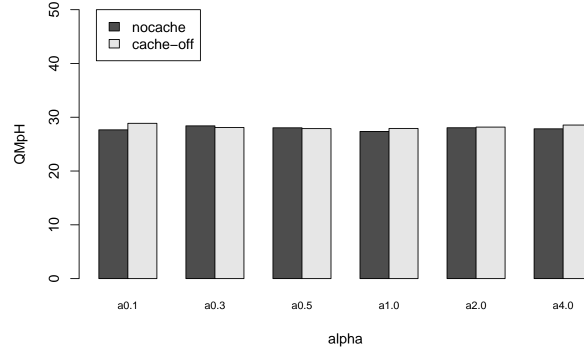


Fig. 4: Cost of Caching, BSBM Explore and Update Use Case

products, reviews, and offers being added to, and old offers being removed from the dataset. This use case tests performance gains from caching on both static datasets (intra-query mix) and updated dataset (inter-query mix, after an update set). The updates contain both relevant and irrelevant data to the queries in the Explore use case.

Figure 3 shows the performance improvement of our caching system on the BSBM Explore and Update use case, again with α distribution values ranging from 0.1 to 4.0, and shows between 2–160% increase in benchmark performance.

5.4 Cost of Caching

To evaluate whether implementing our caching system increases overall processing cost, we evaluated the difference in performance between two versions of our system. In one version ("nocache"), we compile our system without any cache-supporting code. In the other ("cache-off"), caching support is included. However, both of these versions were tested against the Explore and Update use case with no caching support enabled in the test driver. In the cache-enabled version, this tests both the cost of cache probing to generate the `Last-Modified` response header during the explore phase and the cost of maintaining mtimes during the update phase. As can be seen in figure 4, the cache-enabled system performs roughly the same as the version without caching (executing at times both slightly faster and slower than the baseline "nocache" system).

5.5 Discussion

The performance of this system is not competitive with existing SPARQL stores, which achieve dramatically higher scores on the Berlin SPARQL benchmark. We attribute this to our system being a testing implementation meant to demonstrate our cache-supporting indexes and algorithms, not meant to compete head

to head with production systems. Specifically, our system uses a very basic B+ tree implementation with no optimization to reduce disk IO, and lacks any query optimizer or memory management of database pages. We would have liked to evaluate our caching approach using a more efficient implementation, but found that modifying the low-level index structures with mtime fields *and* making those fields available through many layers of API abstractions was very difficult. Overall, we suggest attention should be paid to the large relative improvement of performance with query result caching, not on the specific QMpH figure of our implementation.

We believe our system's lack of database page management may hurt overall caching performance. The ability to cache database pages in memory would not only improve overall performance, but in some cases would specifically improve performance of the cache query (probe) algorithm. Specifically, in cases where the upper levels of index nodes reside entirely in memory, accessing the LCA nodes that provide the mtime of relevant data may require no disk access whatsoever.

Conversely, a highly optimized system tuned for very fast pattern matching and joins would narrow the performance gap between validating a conditional query with cache probing and evaluating the query in full. This situation would seem to provide less benefit from caching. However, this narrowing of performance gap is only one aspect of the benefits from caching. Even on a very efficient implementation, caching would still reduce the network IO required to transfer the query results (which our evaluation does not address) and the memory usage on the server.

6 Conclusion

Caching of SPARQL query results is a promising approach to improving scalability. In this paper we have shown that simple modification of the indexing structures commonly used in SPARQL processors can allow fine-grained caching of query results based on the freshness of data relevant to the query. We evaluated this system using the Berlin SPARQL Benchmark and found that caching can dramatically improve performance in the presence of repeated queries. Moreover, maintaining the data required for caching and using it to service conditional query requests has low cost compared to fully evaluating queries.

In the future we hope to apply the presented caching structures and algorithms to an existing, optimized SPARQL processor and evaluate it using much larger datasets and more expressive queries than those provided by BSBM. We also believe this work could be improved in many ways. The precision (and therefore the soundness) of the cache probing algorithm might be improved by taking into account the ways in which relevant data is combined and modified (e.g. using joins, filters, and projection). In many cases, typical queries use only a subset of available database indexes. Cache-enabled search trees for only the most frequent access patterns could be augmented with non-tree indexes, allowing the system to leverage the benefits of certain non-tree index structures while keeping the precision of caching for frequent queries. Finally, other index-

ing structures might also be modified to store similar fine-grained caching data, allowing informed indexing structure choices while maintaining the benefits of caching.

Acknowledgements We thank Timothy Lebo and Lee Feigenbaum for their helpful comments and suggestions about this work.

References

1. Breslau, L., Cao, P., Fan, L., Phillips, G., Shenker, S.: Web caching and Zipf-like distributions: Evidence and implications. Proceedings of INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (1999)
2. Gallego, M., Fernández, J., Martínez-Prieto, M., Fuente, P.: An empirical study of real-world SPARQL queries. USEWOD2011 - 1st International Workshop on Usage Analysis and the Web of Data (2011)
3. Harth, A., Decker, S.: Optimized index structures for querying RDF from the web. Proceedings of the 3rd Latin American Web Congress (2005)
4. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proceedings of the VLDB Endowment archive (2008)
5. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for rdf. Proceedings of the VLDB Endowment archive (2008)
6. Harris, S., Lamb, N., Shadbolt, N.: 4store: The design and implementation of a clustered rdf store. Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009) (2009)
7. Goldstein, J., Larson, P.: Optimizing queries using materialized views: a practical, scalable solution. Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (2001)
8. Amiri, K., Park, S., Tewari, R., Padmanabhan, S.: Dbproxy: A dynamic data cache for web applications. Proceedings of the 19th International Conference on Data Engineering (ICDE'03) (2003)
9. Larson, P., Goldstein, J., Zhou, J.: Mtcache: transparent mid-tier database caching in sql server. In: Proceedings. 20th International Conference on Data Engineering. (2004) 177 – 188
10. Martin, M., Unbehauen, J., Auer, S.: Improving the performance of semantic web applications with SPARQL query caching. Proceedings of the 7th Extended Semantic Web Conference (ESWC) (2010)
11. Hartig, O.: How caching improves efficiency and result completeness for querying linked data. Proceedings of the 4th Linked Data on the Web (LDOW) Workshop (Mar 2011)
12. Guéret, C., Groth, P., Oren, E., Schlobach, S.: eRDF: A scalable framework for querying the web of data. (Oct 2010) 1–17