Graph-Aware, Workload-Adaptive SPARQL Query Caching

Nikolaos Papailiou[⋆] Dimitrios Tsoumakos[†] Panagiotis Karras[◊] Nectarios Koziris[⋆]

[⋆]National Technical University of Athens, Greece
{npapa, nkoziris}@cslab.ece.ntua.gr

[†]Ionian University, Greece
dtsouma@ionio.gr

[◊]Skoltech, Russia
karras@skoltech.ru

ABSTRACT

The pace at which data is described, queried and exchanged using the RDF specification has been ever increasing with the proliferation of Semantic Web. Minimizing SPARQL query response times has been an open issue for the plethora of RDF stores, yet SPARQL result caching techniques have not been extensively utilized. In this work we present a novel system that addresses graph-based, workload-adaptive indexing of large RDF graphs by caching SPARQL query results. At the heart of the system lies a SPARQL query canonical labelling algorithm that is used to uniquely index and reference SPARQL query graphs as well as their isomorphic forms. We integrate our canonical labelling algorithm with a dynamic programming planner in order to generate the optimal join execution plan, examining the utilization of both primitive triple indexes and cached query results. By monitoring cache requests, our system is able to identify and cache SPARQL queries that, even if not explicitly issued, greatly reduce the average response time of a workload. The proposed cache is modular in design, allowing integration with different RDF stores. Incorporating it to an open-source, distributed RDF engine that handles large scale RDF datasets, we prove that workload-adaptive caching can reduce average response times by up to two orders of magnitude and offer interactive response times for complex workloads and huge RDF datasets.

1. INTRODUCTION

The RDF standard [6] together with the SPARQL[8] query language have been acknowledged as the de facto technologies to represent and query resources in the Semantic Web era. The schema-free nature of RDF data allows the formation of a common data framework that facilitates the integration of data originating from different application, enterprise, and community boundaries. This property has led to an unprecedented increase in the rate at which RDF data is created, stored and queried, even outside the purely academic realm (e.g., [9, 2]) and consequently to the development of many RDF stores [37, 29, 40, 11, 30, 12] that target RDF indexing and high SPARQL querying performance.

However, the move from the schema-dependent SQL data to the schema-free RDF data has introduced new indexing and querying challenges and made a lot of the well-known relational database optimizations unusable. In fact, RDF databases assume limited knowledge of the data structure mainly in the form of RDFS triples [5]. However, RDFS information is not as rich and obligatory as the SQL schema; it can be incomplete and change rapidly along with the dataset. Therefore, most RDF databases are targeting the indexing of individual RDF edges resulting in query executions with much more joins than processing the same dataset in a schema-aware relational database. In contrast, RDF databases that use RDFS information to store and group RDF data [34, 35], fail to effectively adapt to schema changes and to non-conforming, to the schema, data. In general, RDF databases have not yet effectively benefited from the classic schema-aware optimizations used in SQL databases:
• Grouping data that are accessed together using tables.
• Indexing according to filtering and join operations.
• View materialization of frequently queried data patterns.
We argue that all those optimizations can be employed by an RDF database, without any prior knowledge of both the data schema and the workload, by actively monitoring query requests and adapting to the workload.

Result caching is a methodology that has been successfully employed over different applications and computing areas to boost performance and provide scalability. Given the complexity and very high execution latencies [29, 30] of several SPARQL query patterns, caching of frequent RDF subgraphs has the potential of boosting performance by even orders of magnitude. While indexing of graph patterns is extensively used in state of the art graph databases [41, 38], RDF stores have not yet taken advantage of these techniques. What is more, these schemes focus on *static* indexing of important graph patterns, namely they index subgraphs based solely on the underlying dataset, without any regard for the workload. However, the diversity of the applied SPARQL workloads together with the requirement for high performance for all the different workloads calls for dynamic, workload-driven indexing solutions (e.g., [18]).

In this work we argue for a *workload-adaptive* RDF caching engine that manages to dynamically index frequent workload subgraphs in real time, and utilize them to decrease response times. The major contributions of this paper are:
• A SPARQL query *canonical labelling* algorithm that is able to generate canonical string labels, identical among all

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright © 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2723714.

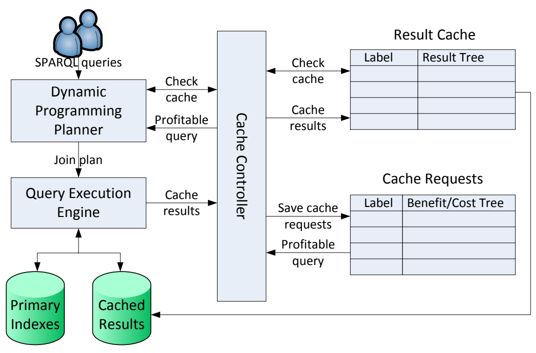

Figure 1: System architecture

isomorphic forms of a SPARQL query. We use these labels to identify and request query graphs to or from the cache. Most importantly, this scheme enables unique identification of all common subgraphs inside any SPARQL workload.

• We integrate our SPARQL query *canonical labelling* algorithm with a state-of-the-art *Dynamic Programming Planner* [28] by issuing cache requests for all query subgraphs using their canonical label. The resulting optimal execution plan may thus involve, in part or in whole, cached query subgraphs. In addition, we not only examine the utilization of exact cached subgraphs, but also larger, more general graphs that can be used to provide the input of a query subgraph.

• A *Cache Controller* process complements the caching framework. The controller monitors all cache requests issued by the workload queries, enabling it to detect cross-query *profitable* subgraphs and trigger their execution and caching.

Our caching framework is modularly designed to be able to support different RDF processing engines and their respective indexing and cost models. It specifically targets disk-based caching of RDF graphs for read-only workloads. Persistent storage can support immense volumes of cached results, that can be also indexed according to the engine's indexing capabilities. We integrate our prototype implementation with H_2RDF+ [30], a state of the art distributed query execution engine that utilizes multiple RDF indexes along with efficient distributed merge and sort-merge joins. Utilizing HBase and HDFS as the cache storage backends, our system is able to handle arbitrarily large RDF graphs. Extensive evaluation results using diverse workloads show that our caching framework is able to effectively detect, cache and utilize SPARQL query results to offer interactive, millisecond-range average response times for several workloads, reducing them up to two orders of magnitude.

2. SOLUTION OVERVIEW

The goal of this work is to provide an efficient, *workload-based* caching system that can adapt cache contents according to the query requests, trying to minimize response times for any SPARQL workload.

As depicted in Figure 1, query resolution starts from the *Dynamic Programming Planner* (Section 4) that identifies the optimal execution plan. We adapt a state-of-the-art dynamic programming planner [27], in order to efficiently discover and utilize all possibly usable cached query patterns that relate to the query in hand. To achieve robust tagging of query graphs in the face of multiple *isomorphs*, we employ a novel *canonical SPARQL query labelling* algorithm (Section 3) that allow us to uniquely index and refer to SPARQL query patterns. Meta-data about cached results are stored in-memory, indexed using their canonical labels, in the *Result Cache* table. To compute the optimal cost for each SPARQL query, our planner iterates over all its subgraphs issuing cache checks using their canonical label. The benefit of utilizing cached results is evaluated by our planner using the execution engine's cost model, resulting in query execution plans that can contain operators over both RDF primary indexes (e.g., *spo* tables) and cached results.

To enhance the cache hit ratio as well as decrease query response times, we not only examine exact query subgraphs but also search for more general cached results that can provide input for a query subgraph by executing a filtering operation over one or several of its variables. Our *Cache Controller* module (see Section 5) is responsible for accessing the *Result Cache*, as well as for recording all cache requests using the *Cache Requests* table. During query execution, all intermediate and final results are considered for caching according to the specified result caching strategy. The *Cache Controller* is responsible for monitoring cache requests and maintaining detailed benefit estimations for possibly usable query patterns as well as for their materialization cost. Utilizing this information, we trigger the execution and caching of *profitable* queries (frequently requested but not cached queries) in order to boost the cache utilization. The resulting architecture is pictorially described in Figure 1.

3. SPARQL QUERY CANONICAL LABELLING

In this section we address the problem of efficiently indexing SPARQL query results. Indexing graph patterns is a challenging task because it requires to tackle the graph isomorphism problem [19], a fundamental problem in graph theory. This problem arises when the same query pattern appears in different queries with small deviations such as pattern reordering, variable renaming etc. For example the following SPARQL queries are isomorphic.

```
?p ub:worksFor "MIT" .              | ?v1 ub:name ?v2 .
?p ub:name ?name .                  | ?v1 ub:emailAddress ?v3 .
?p ub:emailAddress ?email .         | ?v1 ub:worksFor "MIT"
```

All "isomorphs" of the same SPARQL graph must be identified and linked to the same cache entry for a graph caching scheme to work. To address this issue, we extend a solution for graph canonical labelling and introduce the concept of SPARQL graph canonical labelling.

Definition 1. *A graph labelling algorithm C takes as input a graph G and produces a unique label L=C(G). C is a canonical graph labelling algorithm if and only if for every graph H which is isomorphic to G we have C(G)=C(H). We call L a canonical label of G. Additionally, L introduces a canonical total ordering between the vertices of G.*

The canonical labelling problem shares the same computational complexity with the graph isomorphism problem and belongs to the GI complexity class. GI is one of the few open complexity classes that is not known to be either polynomial-time solvable, or NP-complete[19, 15]. To date, there exists a lot of heuristic evidence that GI is not NP-complete and there are many efficient open-source implementations for both graph isomorphism and canonical labelling algorithms [4, 1, 7] that are able to handle really large graphs. One of the first and most powerful canonical labelling algorithms is McKay's *nauty* algorithm [24] that introduced an innovative use of automorphisms to prune the isomorphism search space. In this paper we select to use *Bliss* [1] for computing graph canonical labels. *Bliss* extends *nauty* by introducing some extra heuristics that boost its performance on difficult graphs. Furthermore, it offers a very efficient C++ library that can compute canonical labels for graphs with thousands of vertices in milliseconds making it ideal to handle even the most complex SPARQL query graphs. The most descriptive format that *Bliss* and most of the above algorithms work with is the directed vertex-colored graph.

Definition 2. *A directed vertex-colored graph is a graph G=(V,E,c), where $V = \{1, 2, ..., n\}$ is a vertex set, $E \subseteq V \times V$ is a set of directed edges, and $c : V \to N$ is a function that associates to each vertex a non-negative integer(color).*

In order to use the existing canonical labelling algorithms, we need to transform SPARQL queries to directed vertex-colored graphs without losing any information that can introduce false positives or false negatives, i.e., non-isomorphic SPARQL queries having the same label or isomorphic queries having different labels. Our first task is to transform SPARQL queries to directed vertex-and-edge-colored graphs.

Definition 3. *A directed vertex-and-edge-colored graph is a graph G=(V,E,c_v,c_e), where $V = \{1, 2, ..., n\}$ is a finite set of vertices, $E \subseteq V \times V$ is a set of directed edges, $c_v : V \to N$ and $c_e : E \to N$ are color functions for vertices and edges.*

As a running example, let us assume that we need to get a canonical label for the following SPARQL query:

```
?prof ub:teacherOf ?gcourse .  (1)
?prof ub:teacherOf ?ugcourse .  (2)
?ugcourse rdf:type ub:UndergraduateCourse .  (3)
?gcourse rdf:type ub:GraduateCourse .  (4)
```

Initially, we transform the query to a graph where each triple query is represented by a vertex and triple queries that share a common variable are linked with an edge. This graph is depicted on the left of Figure 2, where the vertex IDs correspond to the triple query labels presented above. In the next stage, we remove all information related to the variable names. To do so, for each triple query we create a label that consists of three IDs, one for each position inside the triple. Bound nodes are translated to integer IDs using a String to ID dictionary while variables are translated to zero. For the current example, let us assume that the String-ID dictionary contains: {teacherOf→15, type→3, UndergraduateCourse→52, GraduateCourse→35}. The label of the first triple query would be, for example, "0_15_0". We handle *OPTIONAL* triple query patterns by appending a special character '|' at the beginning of their label. We also direct and color the edges according to the type of join that they represent. All possible types are {SS, SP, SO, PS, PP, PO, OS, OP, OO}. As we will see later in this section, the number of edge colors affects the labelling performance so we need to reduce it as much as possible. To do so, we introduce a position ordering (S<P<O) and only add edges whose source position is lower than or equal to their destination position. We only require the following 6 edge types {SS, SP, SO, PP, PO, OO}. In the second graph of Figure 2, we can see both the vertex and edge labels for our query.

In the final step, we translate the vertex and edge labels to non-negative integers(colors). We just sort the vertex labels and use as color the position of the label in the sorted set. For edge labels we use the following translation {SS→1, SP→2, SO→3, PP→4, PO→5, OO→6}. The final graph is a directed vertex-and-edge-colored graph and can be seen in the first graph of Figure 3. We note here that we do not use

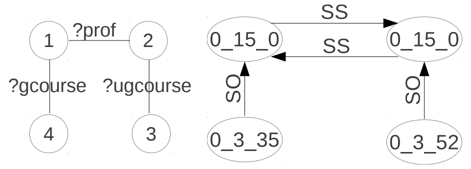

Figure 2: SPARQL query transformation

grouping, ordering, filtering and projection information for labelling SPARQL queries. Queries that are isomorphic but differ for example in result ordering will get the same label but they will be handled as different versions of the same result by our dynamic planner described in Section 4.

The above transformation removes all information relevant to variable naming while managing to maintain all structural information of the query using the directed join edges. To prove this, we note that isomorphic queries that contain different variable names would generate the same labelled graph because variable names do not affect the labels. Respectively, we can generate an isomorphic SPARQL query from the transformed graph by assigning variable names. We initially give unique names to all variables, labelled as 0. We then iterate over all edges, equating their source and destination variable names using the edge label information. This process generates an isomorphic query proving that our transformation maintains all structural query information.

In [25], McKay presents a practical way to transform directed edge-colored graphs to simple directed graphs without changing their automorphism group. We use this technique to transform directed vertex-and-edge-colored graphs to directed vertex-colored graphs. More specifically, if there are v vertex colors and all available edge colors are integers in $\{1, 2, \ldots, 2^d - 1\}$, we construct a graph with d layers, each of which contains n vertices, where n is the number of vertices of the initial graph. We require only 6 edge colors, one for each possible type of triple pattern join, and thus d equals to 3 leading to a new graph that contains $3n$ vertices. We vertically connect the vertices of the different layers (each corresponding to one vertex of the original graph) using paths. The vertex colors of the first layer remain the same as in the original graph and they get propagated to higher layers by adding v to the corresponding color of the lower layer. For each edge of the original graph the binary expansion of its color number tells us which layers should contain the horizontal edge. For example, an edge with color 3, whose binary expansion is 011, will be placed both in the first and the second layer of the new graph. The transformation for the above example is depicted in the second graph of Figure 3, where the vertex IDs represent the assigned colors.

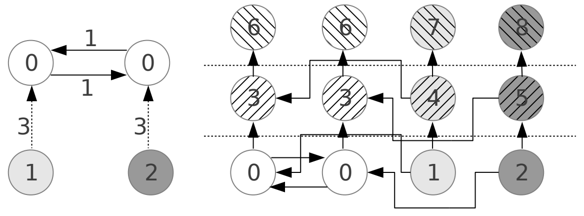

Figure 3: Transformation to directed vertex-colored graph

At this point we can use *Bliss* to produce a canonical label for the above graph. The canonization process returns a canonical order of the graph's vertices. As stated in [25], the order by which a canonical labelling of the new graph labels the vertices of the first layer can be taken to be a canonical labelling order of the original graph. Thus, after executing *Bliss* we can get a canonical ordering for our initial query. For our example the canonical ordering is {1,2,4,3}, where the ids are the SPARQL triple query ids used in the initial graph. Using this ordering, we generate the string canonical label of the SPARQL query graph. We iterate through the triple queries using this canonical ordering and, for each triple query we append its signature at the end of the label string. While iterating, we also generate the canonical ordering of the variables, i.e. variable ?prof is the first variable that we find and thus it gets the canonical ID ?1. In our example, the canonical variable mapping is {?prof→?1, ?gcourse→?2, ?ugcourse→?3} and the generated string label is ?1_15_?2&?1_15_?3&?2_3_35&?3_3_52. Our algorithm produces the exact same string label for any isomorphic SPARQL query and thus is a canonical labelling algorithm for SPARQL query graphs. This label is used as key in both our Result Cache and Cache Requests tables and provides a robust and efficient way to store and retrieve information about SPARQL query graphs.

4. QUERY PLANNING

Finding the optimal join plan for complex queries has always been a major research challenge in optimizing database systems. In addition, our system needs to effectively discover which of the maintained cached results can be used to provide input for a query's subgraph. Both these tasks have exponential complexity to the size of the query because they need to at least check all its subgraphs. While there exist several greedy, heuristic approaches for SPARQL query planning [36, 30], they cannot be easily integrated with a cached result discovery algorithm that finds *all* relevant cached results. In contrast, dynamic programming query planning approaches [27, 29] explore all subgraphs of a query and thus can be easily modified to achieve both optimal query planning and cached result discovery.

One of the oldest and most efficient dynamic programming algorithms for join planning is *DPsize* [13] that is widely used in commercial databases like IBM's DB2. *DPsize* limits the search space to left-deep trees and generates plans in increasing order of size. A more recent approach, *DPccp* [27] and its variant *DPhyp* [28] are considered to be the most efficient, state-of-the-art dynamic programming algorithms for query optimization. They reduce the search space by examining all connected subgraphs of the query in a bottom-up fashion. In addition, *DPccp* is successfully utilized to generate optimal SPARQL join plans in RDF-3X [29]. In this paper, we extend the *DPccp* algorithm and add support for cached result discovery for all query subgraphs.

4.1 Cached result discovery

In this section, we describe how our dynamic programming planner discovers all cached results that can enhance the execution of the query in hand. Detailed descriptions and pseudocodes of the proposed algorithms can be found in Appendix C. The main idea is that while the planner examines all the connected subgraphs of the query, it generates for each one a canonical label and issues a cache check. The benefit of all discovered cached results is examined by the cost model during the planner's execution. The above approach locates all cached results that exactly match a subgraph of the query and evaluates their usability for the generation of the optimal query plan. However, it cannot discover results that can provide input for a subgraph by executing a filtering operation over one or several of their variables. Lets examine the following query:

```
?prof ub:worksFor "MIT" .  ?prof ub:emailAddress ?email .
?prof ub:name "Mike" .
```

Using exact matching, our planner would only issue the cache requests that contain all the bound literals and URIs of the initial query, depicted in Figure 4.

?prof ub:worksFor "MIT" . ?prof ub:name "Mike" .	?prof ub:worksFor "MIT" . ?prof ub:emailAddress ?email .
?prof ub:emailAddress ?email . ?prof ub:name "Mike" .	?prof ub:worksFor "MIT" . ?prof ub:emailAddress ?email . ?prof ub:name "Mike" .

Figure 4: Exact cache requests

However, we notice that the following indexed cached result, while not examined, could also be beneficial for answering the query, transforming it to a simple index lookup:

```
{ ?prof ub:worksFor ?univ .  ?prof ub:emailAddress ?email
?prof ub:name ?name } index by ?univ ?name
```

To address this scenario, we need to also examine more general graphs as well as their indexing properties that can help reduce the filtering operation overhead. For example and in the above case, caching the general result without any indexing might be useless, if for instance its size was significantly larger than the size of the filtered results, as we should read all its data in order to find the relevant ones. In addition, the usability of a cached result also depends on the join operation that must be applied on it. For example, if we need to join this result with another triple pattern according to variable `?prof`, we would like to have it sorted in order to perform a merge join operation and not a more complex and inefficient hash or sort-merge join operation. Furthermore, we need to take into account the case of cached results or queries that contain more complex filters on variables, projection and group-by clauses. To achieve all these goals, we need to find an efficient way to examine which of the cached results can provide input for a query subgraph, due to filtering, projection and grouping properties, and search them to find the one that incurs the least cost.

To tackle this problem, we first abstract our query graph. We remove all bound query nodes that reside in the subject or object position of a triple query and replace them with variables along with the respective equality filters. In the above example, the query would be transformed to:

```
{ ?prof ub:worksFor ?univ .  ?prof ub:emailAddress ?email
?prof ub:name ?name } filter(?univ="MIT", ?name="Mike")
```

We use this abstract query graph in our dynamic programming planner and thus check all its subgraphs issuing cache requests. Each cache request is issued having as key the canonical label of the abstract query subgraph and is accompanied by the filter, projection and group-by clauses of the query as well as by a request for a join variable. As mentioned in Section 3, filters, projections and groupings are not used to generate the canonical label and thus queries with the same abstract structure will be grouped together using their label. The use of canonical labels as cache keys reduces the results that we need to examine for every query subgraph but we still need an efficient way to select the best result from the list of all existing results that share the same abstract structure. Each cache record, related to an abstract query label, maintains all cached results that share this label

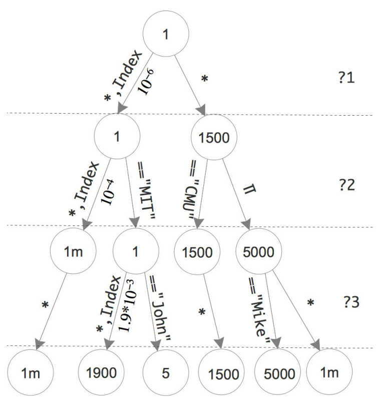

Figure 5: Cached Result Tree

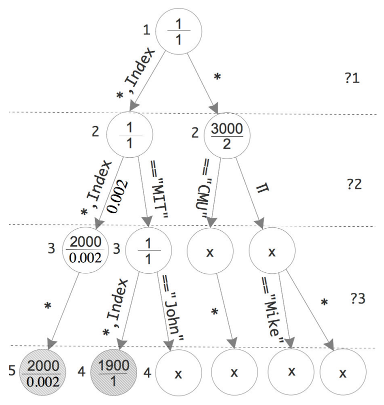

Figure 6: Search Result Tree

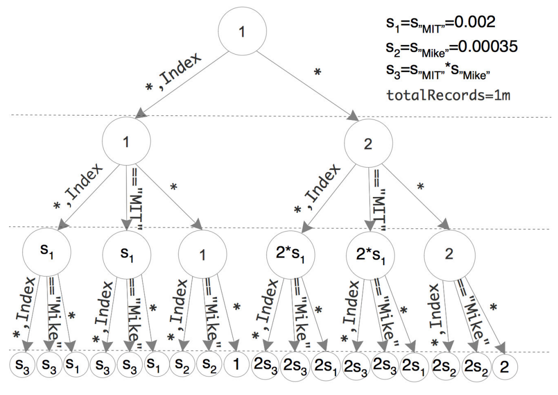
$s_1 = s_{\text{"MIT"}} = 0.002$
$s_2 = s_{\text{"Mike"}} = 0.00035$
$s_3 = s_{\text{"MIT"}} * s_{\text{"Mike"}}$
totalRecords=1m
1
*, Index
*
1
2
*, Index
=="MIT"
*
s_1
s_1
1
$2*s_1$
$2*s_1$
2
*, Index
=="Mike"
*
s_3 s_3 s_1 s_3 s_3 s_1 s_2 s_2 1 $2s_3$ $2s_3$ $2s_1$ $2s_3$ $2s_3$ $2s_1$ $2s_2$ $2s_2$ 2

Figure 7: Benefit Estimation Tree

using a tree structure. This structure can be used to efficiently search for the best result with respect to the request's auxiliary information(filters, projections, groupings). Figure 5 depicts this structure for the following abstract query:
`{ ?prof ub:worksFor ?univ .  ?prof ub:name ?name }`
For this example, we assume that our cache contains several cached results with the same abstract query pattern. As discussed in Section 3, the canonical labelling of the abstract query provides a canonical ordering of its variables. Let's assume that the canonical ordering for our current example is {?prof→?1, ?univ→?2, ?name→?3}. Each level of the tree encodes the information related to the respective variable. Each leaf of the tree represents a cached result and therefore the path that connects it with the root node encodes all its auxiliary information. For example, an edge with label *"*, Index"* means that the result contains no filter on this variable and provides an index for it. An edge labeled *=="MIT"* appearing at the second level of the tree denotes that the result contains the filter *?2="MIT"*. A "Π" edge denotes that the respective variable is projected out in the result. If the query contains a group-by clause it is encoded as a last level edge in the cached result tree. To check the usability of cached results for a cache request we can traverse the tree from the root node and find the results that can be utilised to answer it by following only the edges that provide more generic results than the query request.

Apart from checking the usability of cached results for a cache request, we also need to be able to evaluate their cost and find the result that best matches each cache request. As cost of a cached result we refer to the amount of records that need to be accessed for using it. To estimate this cost in the presence of several indexed variables and query filters, we need to extend our tree structure with selectivity estimations and result sizes. The procedure of inserting a cached result in the result tree can be seen in Algorithm 2. The *treeInsert* method recursively adds a new leaf node, representing the current result, along with its size in number of records. The *minResults* value of each tree node, visible inside each node of Figure 5, represents an estimation of the smallest cost that can be achieved by the results of its respective subtree. Having guarantees for the smallest cost of a subtree, we can perform the search for a cache request using efficient A* search and prune entire subtrees that do not contain relevant results. To create the minimum cost estimations, each leaf node contains the actual result size; we propagate this value to the parent nodes by keeping each time the minimum value among all children. In the case of indexed edges, we apply the selectivity estimations before we propagate the value from the child to the parent node. If a request contains a filter on a variable that is indexed, we need to reduce the cost of the result by the expected selectivity of the filtering operation on the index. Edges that contain no indexes do not change the cost of the result because we would need to access all their data to perform the filtering operation.

Our cache implementation does not depend on the specific way that an RDF execution engine handles selectivity estimations. To create the cached result tree we only require the maximum selectivity that can be achieved by doing a filtering operation on a certain index. For each indexed edge of a cached result we generate a selectivity value $s = minRecords/totalRecords$, depicted along the indexed edges of Figure 5. In our running example, an index on the general result(1 million records), on variable `?2 (?univ)` has a minimum amount of 100 records and thus its maximum selectivity is $s = 100/1m = 10^{-4}$. To handle the estimation of multiple filtering operations on different variables we follow the independency assumption i.e. the selectivity of two filtering operations with selectivities s_1 and s_2 is $s = s_1 \cdot s_2$. This property allows us to follow paths of filtering operations on the cached result tree and maintain a total estimation by just multiplying the individual selectivities while crossing indexed edges. The *treeInsert* method (Algorithm 2) starts by adding the respective leaf along with its amount of records. We then move from the leaf to the root node and update the minimum values of the parent nodes. When crossing an indexed edge we apply the maximum selectivity by multiplying it with the child's number of records. We also set a minimum value of 1 for the costs of nodes. The complexity of the add operation is linear to the amount of variables contained in the abstract query pattern.

Figure 6 depicts the execution of the *searchTree* method (Algorithm 1) when searching for the cache request:
```
{ ?prof ub:worksFor ?univ .  ?prof ub:name ?name
} filter(?univ="MIT"), joinVar(?prof)
```
As mentioned before, we use the minimum cost estimations of the tree nodes in order to perform an A* search and prune subtrees with large costs. The search procedure starts from the root node and checks all nodes in a best first search according to cost estimations. In Figure 6, each node contains two numbers; its estimated cost (upper number) and its selectivity(lower number). Only edges that can be used for answering the request are followed. For example, the edge with label *?3="John"* will not be followed because it is more restrictive than the request. Furthermore, when cross-

ing indexed edges that match with a filtering operation of the request we need to apply their selectivity. For example, when crossing the edge *(?2: "*,Indexed")* we need to apply the selectivity of the filter *?2="MIT"*. To do so, we consult the selectivity estimator of the abstract result for the respective variable. We use the abstract result to estimate selectivities because a tree edge can belong to several results with different sizes and we want to have an estimation for the maximum selectivity. In this case the selectivity of *?2="MIT"* is 0.002 because it is expected to return 2000 records and the abstract result contains 1 million records. To estimate the selectivity of several consecutive filtering operations we maintain a selectivity estimation for each open node and propagate it to children nodes by multiplying it when crossing indexed edges with filter operations. Additionally, when crossing join variable edges we multiply the selectivity by 2 if the edge is not indexed due to the fact that the overhead of performing a hash or sort-merge join algorithm instead of a merge join can be approximated by the time to read the input data twice [29]. The minimum cost estimation for each node (upper value) is computed by multiplying its selectivity with the *minResults* estimation depicted in Figure 5.

Our cost estimations can deviate from the actual costs due to the use of the abstract result selectivity estimator and our assumption of independency for filtering operations. Therefore, we perform a top-k search for results and then further examine the cost of each result, inside *DPccp*, using the detailed cost model of the execution engine. In Figure 6, we perform a top-2 search depicting on the side of each node the step in which it was opened. Nodes marked with an $\times$ were not opened; grey coloured nodes depict the results. We observe that we only required 5 steps to find the top-2 cached results and our search managed to prune a large part of the search tree due to auxiliary info mismatches and minimum cost estimations.

5. CACHE CONTROLLER

In this section, we describe the functionality of our Cache Controller module. Its main challenges are the generation and caching of *profitable* query patterns and the cache replacement strategy. As *profitable* query patterns we define queries that, even if not exactly issued by users or executed as intermediate results, could benefit the execution of the workload if cached. For example, consider a workload of queries having the same abstract query structure but random bound selective IDs. Caching only the intermediate results of those queries would not offer a great benefit to the workload because it would achieve cache hits only for queries that share the exact same selective IDs. In contrast, caching the abstract query pattern indexed according to the variable that contains the selective IDs would introduce benefits for all queries, in this special type of workload, transforming them to index scans. Apart from identifying abstract results and their indexing, our cache controller can intelligently identify cross-query frequent subgraphs and index them according to: 1) Their most common filtering variables and 2) their most common join variables.

5.1 Generation of profitable cached results

As discussed in the previous section, our dynamic programming planner issues cache requests for all subgraphs of the abstract query. In addition, these cache requests are issued while optimizing the query plan and can thus be recorded along with estimations about their effect to the execution time of the respective query. Maintaining such a detailed log of cache requests provides valuable information about which query patterns can provide the most benefit to the workload. The exact benefit computation of all cached requests $Q_i = (V_i, E_i)$ to the execution of a query $Q = (V, E)$ would require the execution of *DPccp* for each one of them. To avoid this, we use the following function, that multiplies the query's cost by the fraction of triple patterns covered by the subgraph, to compute a benefit estimation.

$$B(Q_i|Q) = \frac{|V_i|}{|V|} \cdot cost(Q) \qquad (1)$$

This benefit estimation requires the optimal cost of the query Q and can be computed at the end of our planning process. Therefore, during the planning process our planner records all non satisfied cache requests (Q_i) and at the end compiles a list of requests along with their respective benefit and sends it to the Cache Controller for further processing. This means that the complexity cost of attributing benefits to possibly usable query patterns does not affect the execution and planning time of queries because it is done in an offline manner by the separate thread of the Cache Controller module. The Controller maintains a Cache Requests structure containing request benefits indexed by their abstract canonical label. Each record holds a tree structure encoding benefit estimations for requests sharing the same label. Figure 7 depicts this benefit estimation tree, generated by the *addBenefit* method (Algorithm 3), for the following cache request (Q_i) with benefit $B = 3sec$:

```
{ ?prof ub:worksFor ?univ .  ?prof ub:name ?name }
filter(?univ="MIT", ?name="Mike"), joinVar(?prof)
```

Each leaf of the tree represents a query pattern that can possibly benefit the cache request. To attribute benefits to all possibly usable results at each level of the tree we at least follow the *"*, Index"* and the *"*"* edges. If the respective variable contains a filter, we also add an edge containing it. Furthermore, if the record of the Cache Requests table already contained benefits for results with filters that can be used for the current request, they are also followed. For example, if a previous query had a regular expression filter $?name = "M*"$ we would also attribute benefits to its subtree. The values inside the nodes represent the selectivity estimations for the respective pattern. To generate these estimations we only require a selectivity estimator of the abstract result and the total amount of records of the abstract result. We again use the independency property to estimate the selectivities for multiple filters on different variables. Therefore, we just need to propagate the selectivity estimation from the parent to the child node by multiplying with the selectivity of the respective edge. The benefit for each usable result, leaf node, is:

$$b = B - s \cdot R/thr \qquad (2)$$

where B is the total benefit of the request, s is the selectivity of the result, R is the number of records of the abstract query, and thr is the engine's read throughput (e.g. 100k records/sec). The second part of the equation represents the cost to read the result. In Figure 7, the benefit for a result with selectivity s_3 is $b_3 = 3sec - s_3 * 1m/100k \simeq 3sec$. We also ignore benefits that are less than 0, for example the non indexed abstract result, rightmost leaf, has selectivity 2 and benefit $b = 3sec - 2 * 1m/100k = -17sec$. When the benefits of all patterns are estimated, the Cache Controller sums the previously existing benefit values with the new ones and stores the tree inside the record of the Cache Results table. The Controller, running our planner, estimates the execution cost of the most prominent requests and maintains an ordered list of (request, benefit/cost) pairs (Algorithm 5) used by the *profitableQueryGeneration* method to trigger the caching of the most *profitable* queries.

The query pattern tree grows exponentially to the number of variables in an abstract query and there are various computed and maintained benefits for results that will never be the most profitable. However, this exponential complexity does not affect our query response times because it is done independently of the query execution. To alleviate the problem of maintaining all the benefit estimations, we have an offline process (Algorithm 5) that runs in configurable timeframes (e.g. every 10sec or every 10 queries) and maintains only the top-k leaf nodes of the Cache Requests table. Furthermore, to avoid promoting only queries with large costs, we normalize the benefit estimation of a query pattern using its execution cost. Thus, in the previous example, if we had queries that mostly targeted "MIT", both the indexed version of the abstract result and the version that contains only records for "MIT" would gather the same benefit but eventually the more specific result would be cached due to its lower cost of execution.

In addition, queries issue cache requests adding benefit to all their subgraphs. When one query pattern gets selected for caching the corresponding benefit should be removed from all other patterns that were requested by the same queries as they were *satisfied*. Not reducing benefits for *satisfied* queries can lead to: 1) executing all subgraphs of a frequent query pattern, 2) difficulties in identifying new profitable cache requests due to obsolete benefit estimations of *satisfied* requests. To alleviate this problem, we maintain for each query pattern in the Cache Requests table a list of query IDs that attributed to its benefit. This list is used to reduce the benefit of cache requests after the execution of a *profitable* query. We reduce the benefit of each request by the fraction of its query IDs that belonged to the *profitable*'s query ID list. In our example above, if we decided to execute the abstract query indexed according to variable $?univ$, all query profits would go to zero because the *profitable* query gathered benefit from all workload queries. Thus, no more queries would be executed by the Controller. To avoid maintaining obsolete benefit estimations for cache requests, we additionally decrease their benefit with time.

5.2 Cache replacement strategy

In this paper, we target disk based cached results keeping only their meta-data in main memory. Thus, the amount of the maintained cached results is only limited by the available disk space. However, the user can set limits on the disk space capacity dedicated for storing cached results. When a new cached result cannot be stored without exceeding those limits we need to remove some of the existing cached results. To be able to intelligently select which result should be evicted from the cache, the Controller also maintains a benefit estimation for each cached result. This benefit estimation is updated using the method *addResultBenefit* (Algorithm 4) and differs from the one described in the previous section. After the execution of a query that used a cached result, the Controller executes the planner one more time, restricting the use of this cached result. This gives us the cost of the query without utilizing the respective cached result. The controller adds the difference between this cost and the actual cost to the result's benefit value. Benefits, also, decrease with time in order to avoid maintaining obsolete results. We prioritize cache evictions using Algorithm 6 that utilizes this benefit estimation. New results can only evict cached results that have less benefit. In addition, if the cache size constraints are violated, our controller will not execute new *profitable* queries unless their benefit is larger than the lowest cached result benefit. To evict multiple results, due to its size, a new result must have benefit greater than the sum of benefits of all evicted results.

6. RELATED WORK

In this section, we present related research on SPARQL result caching techniques as well as on RDF databases, discussing the effect of our caching framework on them.

6.1 SPARQL result caching

Here, we present some of the most relevant research approaches on SPARQL query result caching. While this is a challenging problem, few relevant approaches to address it exist. A first attempt to introduce SPARQL caching was made in [23], where a meta-data relational database is responsible for storing information about cached query results. However, this approach cannot tackle the isomorphism problem introduced when the same SPARQL graph pattern is requested from different queries with small deviations such as pattern reordering, variable renaming, etc.

A more sophisticated approach was presented in [39], where the cache keys consisted of normalized Algebra Expression Trees (AETs) that correspond to cached join plans. However, a cached AET is only used in join plans that exactly contain it as a subtree. This is also not a general subgraph caching framework and leads to inferior cache utilization.

In [22], the authors introduce a similarity-based matching algorithm that can detect cached queries that resemble the query in hand. Yet, this greedy technique cannot find all candidate subgraph matches for a SPARQL query. They also propose some heuristics used to augment the workload queries and prefetch SPARQL query results, which resembles our *profitable* query execution. Their approach is again based on heuristic techniques that can not examine the benefit of all possibly usable results as well as their indexing. To sum up, current caching frameworks fail to effectively locate all subgraph matches and subsequently use them to produce optimal query plans for SPARQL queries.

Apart from SPARQL result caching there are other relevant techniques that try to tackle parts of the problem that we address in this paper. Many of the state-of-the-art approaches in graph database indexing propose the use of frequent pattern indexing [41, 38]. Frequent and discriminative subgraphs are discovered and indexed during the import phase of the dataset and can be then utilized to efficiently answer queries that contain them. In addition, large amount of research has been done in the area of multi-query optimization [33] and view selection [26] for relational databases. Approaches for multi-query optimization have also been proposed for SPARQL query processing [21]. All these techniques depend either on knowledge of the workload queries or on expensive import procedures that find frequent patterns in the dataset. In contrast, our approach assumes no a-priori knowledge on both the dataset and the workload and targets the adaptive indexing and caching of frequent query patterns by actively monitoring the workload queries.

6.2 RDF datastores

One of the first and most widely used ideas for indexing and querying RDF data was presented in Hexastore [37]. According to it, the materialization of all six *permutations* of subject-predicate-object values is required in order to retrieve any triple pattern at minimal cost and extensively use efficient merge joins. A similar indexing approach is followed in RDF-3X [29] which maintains all Hexastore indices as well as additional indices that collect statistical information amounting to a total of 15 indices. RDF-3X also uses the *DPccp* algorithm in order to find the optimal join execution plan and thus our caching framework can be easily integrated and offer the respective performance gains.

BitMat [11] is an alternative approach for storing RDF triples via a 3-dimensional bit matrix. Each matrix element is a bit denoting the presence or absence of the corresponding triple. This 3-d bit matrix is flattened to multiple 2-d matrices used for query execution. BitMat uses a greedy technique to chose the sequence of matrix operations that need to be performed for the execution of a SPARQL query.

To tackle the big-data challenge, research has recently moved onward to distributed RDF data management systems. A first attempt to utilize MapReduce and HBase was made in H_2RDF [31] which materializes three RDF triple indexes using HBase tables. H_2RDF performs joins using the MapReduce framework, employing a greedy planner that decides on the sequence of joins as well as their type of execution ranging from centralized to distributed joins.

An alternative proposal is presented by Huang et al. [17]; this method starts out by partitioning the RDF graph into distinct subgraphs, each stored in a single node running a local RDF-3X instance. Moreover, in a replication scheme, each node keeps information on the graph contents within *n hops* from the contents it owns. This provision allows for unobstructed parallel processing of SPARQL queries satisfying an *n hop guarantee*. In case this guarantee is not satisfied, Hadoop is invoked for distributed join processing.

Lastly, Trinity-RDF [40] is a distributed, memory-based graph engine for RDF data. It stores RDF triples as graph edges using distributed memory hashmaps. Utilizing graph exploration techniques it avoids join execution overheads and takes advantage of the quick random access provided by the memory resident data. Trinity-RDF depends on a DP planner to find the best graph exploration execution plan.

To sum up, all the aforementioned systems utilize different RDF indexing and query execution techniques. They also rely on a range of techniques for query optimization, from dynamic programming planners to greedy heuristic planners. However, none of the above systems effectively utilizes cached results or adaptive indexing techniques. We argue that our caching framework can be integrated with and benefit these systems transparently because:

- The *DPccp* algorithm is generic and can handle the query optimization for all those systems.
- Cost models for the different operators of the various systems can be plugged into the *DPccp* algorithm and used to devise the optimal query execution plan for each engine.
- Our labelling algorithm is based on the SPARQL standard and thus is applicable to all systems.
- Our caching implementation only depends on query metadata and thus query results can be stored and reused using the system's native storage format.

7. EXPERIMENTS

In this section, we present a thorough performance evaluation of our generic RDF caching framework. We choose to integrate our caching framework on top of the open source[1], distributed H_2RDF+ database [30, 32] that combines the Hadoop and HBase to index and query RDF data. H_2RDF+ generates HBase indexes for all permutations of RDF elements, namely *spo, pso, pos, ops, osp* and *sop* [37], along with aggregated statistic indexes that can provide a detailed join cost model. In addition, H_2RDF+ is highly scalable, utilizing MapReduce to perform distributed Merge and Sort-Merge joins. Joins can be executed using either single-machine or distributed jobs according to the join cost resulting in small response times for selective queries. H_2RDF+ utilizes aggressive byte-level compression and result grouping in order to reduce its result space consumption. As we focus on disk-based result caching, aggressive compression is a major plus and boosts the caching efficiency by allowing more results to be maintained.

Our caching framework is independent to the query execution engine and thus only small changes are required for H_2RDF+, as well as for other RDF stores, to support it. Our cache depends on indexing cached results for: 1) filtering operations, 2) boosting performance for join execution. Most RDF databases, including H_2RDF+, utilize order preserving indexes that can be used to efficiently perform both operations. H_2RDF+'s indexed results can be used in both Merge and Sort-Merge joins in the same way as H_2RDF+'s primary triple indexes [30]. Non-indexed results have no ordering properties and are stored in plain HDFS files.

7.1 SPARQL query canonical labelling

In this section we test the performance of our SPARQL query canonical labelling algorithm. Its performance is tightly coupled with *Bliss*'s, with an added transformation overhead introduced when translating SPARQL queries into directed, vertex-colored graphs. To evaluate the efficiency of our algorithm, we measure the time required to canonically label a set of different SPARQL query graphs consisting of paths, stars, cycles and path-stars. Path-stars are graphs that contain a star subgraph and each of the branches of the star consists of a path of two triple patterns.

First, we generate SPARQL query graphs using only one type of triple pattern query, e.g., `?v1 ub:takesCourse ?v2`. This means that all vertexes of the transformed graph, depicted in Figure 2, will have the same label. *Bliss* and *nauty* use vertex color and graph structure information to break the graph symmetry and prune the isomorphism search space when searching for a canonical label. Consequently, for SPARQL queries that contain only one type of triple pattern, our canonical labelling algorithm can benefit only from the graph structure. The response times required to label queries with variable number of triple patterns (of a *single* type) are depicted in the first graph of Figure 8. Path and cycle queries require almost stable labelling time, close to 0.2 ms, due to their small number of isomorphs. In contrast, star queries require a labelling time exponential in the number of triple patterns. Our labelling transformation

[1]http://h2rdf.googlecode.com

generates graphs with edges corresponding to join variables and thus it transforms star SPARQL queries to cliques because all triple patterns are joined according to the central variable of the star. Path-stars are a combination of a star and multiple paths and thus their complexity is higher than that of a simple path with the same size but lower than the complexity of the respective star graphs.

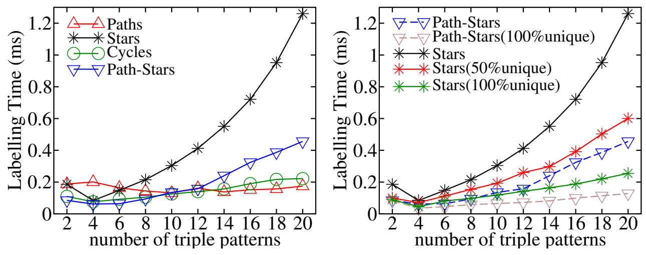

Figure 8: SPARQL query canonical labelling evaluation against queries of variable complexity and size

The second graph of Figure 8 depicts the effect of having different types of triple patterns in the queries. Different triple patterns give our canonical labelling algorithm more information to prune the isomorphism search space. We measure the time needed to label star and path-star queries with variable number of nodes while ranging the percentage of unique triple patterns. For stars we examine queries with 0%, 50% and 100% of the query's patterns being unique while the rest of the patterns are duplicates. We notice that the required labelling time decreases when queries contain more unique patterns that help our algorithm break the graph symmetry. This performance gain is also visible in path-star queries. Finally, this gain is exponential especially in the case of star queries leading to almost linear labelling complexity for both star and path-star queries with 100% unique triple queries. In general, our algorithm is able to label most real-life SPARQL queries in less than 1 ms.

7.2 Dynamic Programming Planner

In this section, we evaluate our dynamic programming planner that integrates *DPccp* [27] with our canonical labelling algorithm. The first graph of Figure 9 depicts the time required for our planner to generate canonical labels for all the query subgraphs, check for existing usable cached results and compute the optimal join plan for several types of SPARQL query graphs. We use a set of query graphs consisting of paths, cycles, stars, path-stars and grids and vary the number of their triple queries. *DPccp* utilizes the edge-graph representation of SPARQL queries, depicted in the left graph of Figure 2. When processing star queries *DPccp* faces an exponential complexity to the amount of triple queries. This is expected because star SPARQL queries are transformed to cliques in the edge-graph and thus contain exponential amount of connected subgraphs that need to be enumerated. However, we can generate optimal plans for star queries with up to 10 triple queries in milliseconds. In contrast, for path and cycle queries that do not contain as many subgraphs, the planner exhibits great performance and is able to process queries with up to 20 relations in less than 30ms. Path-star and grid query graphs, due to their more complex graph structure, present higher complexity than paths and cycles resulting in millisecond response times for queries with up to 14 triple queries.

Another major parameter that affects the complexity of

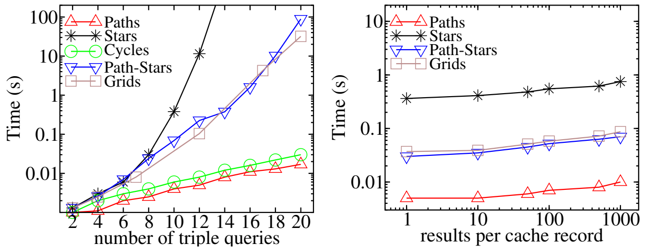

Figure 9: Query planning performance

our planner is the tree search for cached results inside a cache record. This procedure mainly depends on the amount of results that are stored inside a cache record. As explained in Section 4.1, we perform a top-k, A* search to find the best results inside each cache record. From our experimentation, we have discovered that a top-3 search inside each cache record is sufficient to discover the best cached results in all our query examples and therefore we utilize a top-3 search in all the experiments of this section. The second graph of Figure 9, depicts the effect of the amount of cached results to the planners execution time. To test the impact of the amount of cached results to the execution of our planner, we select a set of query graphs all consisting of 10 triple queries and range the number of cached results per cache record, utilized by the results, by randomly generating and loading 1 to 1000 cached results. We assume that 1000 results per unique abstract query subgraph suffice to evaluate our planner's performance even for the most challenging caching scenario. From the experimental results, we note that the performance of our planner scales logarithmically to the amount of cached results for all query patterns. This means that our A* search manages to effectively prune large parts of the cached result trees due to their filtering constraints and our minimum cost estimations. The time needed for our planner to evaluate the same query in the presence of 1 and 1000 cached results per cache record only increases by a factor of 2 for all query patterns. To sum up, our dynamic programming planner is able to both effectively locate cached results and generate the optimal join plan in less than a second for most realistic SPARQL queries; this includes all benchmark queries used in the following sections.

7.3 Cache efficiency

In this section we evaluate the ability of our cache to effectively generate *profitable* results in order to decrease query response times for several diverse SPARQL workloads.

7.3.1 Cluster configuration and Datasets

Our evaluation cluster consists of 10 worker nodes plus a single machine in the role of the Hadoop and HBase master. Each of our workers features a 2 Quad-Core E5405 Intel Xeon®CPUs at 2.00GHz, 8 GB of RAM and a 500GB disk, while the master has similar CPU and disk and 4 GB of RAM. Each worker is set to concurrently run 5 mappers and 5 reducers, each consuming 512MB of RAM. In our experiments, we used Hadoop v1.1.2 and HBase v0.94.5.

We utilize the LUBM dataset generator[14] that creates RDF datasets with academic domain information, enabling a variable number of triples by controlling the number of *university* entities. LUBM is widely used to compare the performance and scalability of triple stores due its ability to create arbitrarily large datasets. We use two datasets in our experiments: LUBM10k (10k universities, 1.38 billion triples and a total of 250GB of data) and LUBM20k (20k universities, 2.8 billion triples and 500GB of data).

7.3.2 SPARQL query workloads

To evaluate caching performance, we use the suite of LUBM read-only benchmark queries [3]. Our caching framework as well as the integrated execution engine (H_2RDF+) offer limited OWL functionality. Thus, we do not consider queries that require complex OWL reasoning in the current evaluation. To better present the advantages of our system we generate 4 representative query workloads:

Selective query workload (W1): This workload consists of queries that contain selective triple patterns, such as `?student ub:takesCourse <Course1>`. These selective patterns can be efficiently used to reduce the size of data processed because they can be directly retrieved using the maintained hexastore triple indexes. H_2RDF+ executes selective queries in a centralized manner resulting to great response times. The LUBM queries for this workload can be found in Appendix A. Our workload issues queries sequentially choosing each time randomly one of the above queries. To make this a more challenging workload, we also randomly alter the bound nodes of each query. For example, LQ1 will be issued each time containing a different graduate course. We can easily generate random IDs because all LUBM URIs follow the format <http://www.Department1.University8.edu/GraduateCourse5>, thus making it easy to generate random department, university and course IDs. The same is true for all IDs present in the aforementioned queries.

Non selective query workload (W2): This workload consists of queries that do not contain any selective triple pattern and require to retrieve and join large relations using distributed joins. Queries in this category have both small and large result sizes because their selectivity is based on graph pattern selectivity and not basic triple-pattern selectivity. Representative LUBM queries from this workload can be found in Appendix A. For example, LQ2 is quite selective because it retrieves the graduate students that are members of the *same* university that they graduated from. While all graduate students have graduated from a university and are members of another university only few of them satisfy the triangle pattern and thus the selectivity of the query is based on the selectivity of this triangular relation. LQ15 produces a large output because it generates information for all universities professors and courses. To make this workload more challenging, we randomly alter the `type` of the triple queries. For example, in LQ9 we randomly generate its first 3 triple queries by alternating between Professor, FullProfessor, AssistantProfessor, AssociateProfessor and Lecturer. In general, there are not many subclasses we can select from and thus the executed workload consists of 19 distinct SPARQL queries.

Subgraph pattern query workload (W3): Another strong point of our caching algorithm is the ability to detect cross-query frequent subgraphs and use them to effectively reduce the workload's response time. To highlight this scenario, we generate a workload of queries that all contain the triangular query patten:

```
?x ub:memberOf ?z .  ?z ub:subOrganizationOf ?y .
?x ub:undergraduateDegreeFrom ?y .
```

Each query in this workload consists of this subgraph along with a set of other triple patterns that are different across the set of queries. Two of the workload queries contain only non selective triple queries like: `?x ub:name ?n, ?x ub:email ?em`, etc. We also add two queries that contain selective patterns like: `?x ub:memberOf <Department0.University0.edu>` and `?z ub:subOrganizationOf <University0>` with random selective IDs. The last query of this workload is the LQ2 which also contains the triangular subgraph along with randomly selected type triple patterns.

General query workload (W4): This workload contains all the queries of the previous workloads. It consists of 14 different SPARQL query types containing both selective and non selective queries. It also contains inter-query subgraph dependencies as discussed above.

7.3.3 Workload caching

In this section we test the caching performance of our system for the aforementioned workloads utilizing the LUBM10k dataset. For each workload, we examine the impact of our cache implementation to the average query response time and highlight the strong and weak points of our system. Figure 10 presents the average query response times for our four query workloads. To highlight the distinct contributions of our system we depict the average response time for: 1) the baseline H_2RDF+ system, 2) our caching implementation with exact cache checks labelled "Exact" and 3) our fully functional system with abstract query cache checks labelled "Abstract". In all cases, we use unlimited cache size by default. We average query response times using a window of 10 queries in order to avoid large randomness in the figures and achieve quick responsiveness to the cache changes. Finally, our Cache Controller is configured to check the Cache Requests table and generate *profitable* queries every 10 sec.

W1: The first graph of Figure 10 illustrates the behaviour of the average query response time for the selective workload. Initially, we note that our baseline H_2RDF+ system has a stable average performance that ranges between 1 and 2.5 seconds. This is expected due to the use of centralized joins that take advantage of HBase indexes. By including the caching mechanism, average response times are gradually decreased to 500ms within 400 sec. Our abstract query cache check technique is especially useful in this workload. Due to the fact that the workload contains random selective IDs, it is true that no exact subgraph of the issued queries could be efficiently used to improve the execution of the consequent queries. This is the reason that the "exact" version behaves in the same way as H_2RDF+. In contrast, when using abstract cache checks our system can effectively discover, execute and index *profitable* SPARQL queries. For example, the controller will cache the following query (Q_c)

```
{ ?x ub:worksFor ?y .  ?x ub:name ?n .
?x ub:emailAddress ?em .  ?x ub:telephone ?t .
?x rdf:type ub:Professor .} index by ?y
```

that can convert the execution of LQ4 from a complex join that takes nearly 2sec, to a simple HBase index access that requires, on average, only several milliseconds. The same holds for all W1 queries, resulting in an average response time of 0.5sec (an average 75% reduction in the mean response time). The response time thus gradually decreases during workload execution as more queries get effectively cached. The 400 sec, required for the response time to reach its lowest point, correspond to the distributed execution and indexing of 6 complex SPARQL queries (like Q_c); yet, their execution does not introduce large overheads to the que-

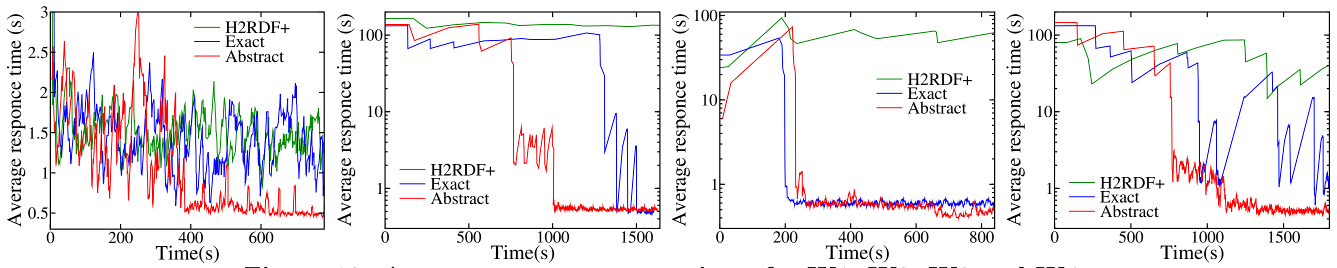

Figure 10: Average query response times for W1, W2, W3 and W4

ries of the workload that run at the same time. Finally, we note that while H_2RDF+'s basic indexes occupy 62GB of disk space, our system requires only an additional 1.3GB of disk space to cache the corresponding results and decrease the average response time for this workload. Although the workload queries are selective, the generated cached results are quite large because they contain information for all their selective nodes. The small size of cached results is mainly attributed to H_2RDF+'s aggressive compression scheme.

W2: The average response times for the compared systems are presented in the second graph of Figure 10. This workload is a lot more execution-intensive than the previous one, leading to an average response time of nearly 130 sec for the plain H_2RDF+ approach. However, our fully functional caching system is able to achieve interactive, millisecond-range response times for this workload in less than 1000 seconds (over two orders of magnitude reduction in the mean response time). Again, our caching system takes advantage of the abstract cache requests technique in order to capture the changes in the type fields of the queries and generate the most profitable cached results. It only requires the execution and indexing of 3 results to offer interactive response times for this workload. For example, the profitable cached result that corresponds to LQ9 is:

```
{?y ub:teacherOf ?z .  ?x ub:advisor ?y .
?x ub:takesCourse ?z .  ?x rdf:type ?t1 .
?y rdf:type ?t2 .  ?z rdf:type ?t3} index by ?t1 ?t2 ?t3
```

The time required to execute these 3 queries is actually around 400 sec. The additional time required to minimize the average response times is introduced by: 1) the fact that the result discovery process will not be very effective until enough entries are gathered in the Cache Requests table, 2)the concurrent distributed execution of workload queries affects the execution time of profitable queries. We notice that the "exact" version of our algorithm can also reduce the average response time for this workload but it needs nearly 1500 sec to do so. As mentioned before, when we do not utilize abstract cache requests we can only benefit from exact subgraph cache matches. However, this workload actually contains only 19 distinct queries and therefore we only need to execute each query once and then all queries will be directly found in the result cache. Despite being execution intensive only one of those queries, LQ15, has large output that requires 5.8GB of disk space. LQ2 and LQ9 are in fact very selective and require not more than 3MB of disk space.

W3: The third graph of Figure 10 depicts the average response times achieved for the subgraph pattern based workload. We can observe that both versions of our caching algorithm are presenting similar average response time behaviours. This is due to the fact that the common triangular query pattern, despite its complexity, is quite selective, with less than 3000 results. Both our algorithms can detect this inter-query dependency and in fact this is the first *profitable* query that is cached in both cases. The selectivity of this pattern is quite large, leading to really efficient, interactive execution times after only 240 sec. After caching the triangular subgraph, our fully functional version will also continue with caching indexed results for all the selective queries of the workload. This procedure leads to another small gain in performance, which can be seen after 600 sec. At this point the results could be directly retrieved using an HBase scan rather than executing a small join, showing over two orders of magnitude reduction in workload's mean response time. The disk size occupied by our cached results in this workload is only 50KB.

W4: The caching efficiency for this workload is depicted in the last graph of Figure 10. We observe that this is a quite challenging workload for our baseline H_2RDF+ engine as it presents an average response time of around 60 sec.

Regarding our caching implementation that only uses exact subgraph matching, we note that it is able to reduce the average response time by an order of magnitude resulting in average response times that oscillate around 3 seconds after 2000 seconds. Our "exact" version is able to cache: 1) the non selective results due to their small distinct number, 2) the frequent subgraph queries by caching their common subgraph. Nevertheless, this caching framework is not able to further reduce the execution time of small selective queries.

Using abstract cache requests we manage to reduce the average response time by two orders of magnitude for this challenging workload, resulting in interactive response times that stabilize close to 500 msec. As mentioned in Section 5.1, our Cache Controller prioritizes the execution of *profitable* queries according to their expected benefit and thus the queries that correspond to the costly, non selective SPARQL queries will be issued first. The large benefits of caching costly queries are obtained early, leading to one order of magnitude better response times after the first 700 sec. After caching large queries, our Cache Controller gradually improves the execution efficiency for the rest of the queries, requiring less than 1300 sec to execute and cache all the relevant, profitable SPARQL queries and drop response times to their lowest values. Due to the aggressive compression of H_2RDF+, the disk space occupied by all the cached results is only 7.5GB, leading us to assume that we can easily scale and handle even more complex datasets and workloads.

7.4 Caching techniques comparisson

To present a more detailed comparison of all different caching techniques for RDF data we use the *Detailed Cost Saving Ratio (DCSR)* metric presented in [20].

$$DCSR = \frac{\sum_i s_i}{\sum_i c_i} \tag{3}$$

where c_i is the execution cost for query q_i without utilising the cache and s_i is the savings provided by using the cache:

$$s_i = \begin{cases} 0, & \text{if } q_i \text{ does not use the cache} \\ c_i, & \text{if there is an exact match for } q_i \\ c_i - cf_i, & \text{if } q_i \text{ uses the cache and has cost } cf_i \end{cases} \quad (4)$$

DCSR captures the different levels of effectiveness of the materialized results against the workload queries and can be used to accurately compare different caching techniques.

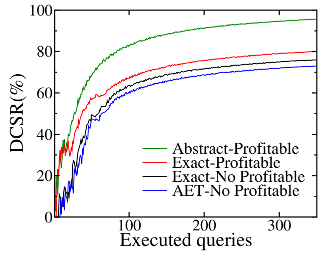

Figure 11: Effectiveness of caching techniques

The $DCSR(\%)$ measurements, for our generic workload **W4**, for four different RDF caching techniques are presented in Figure 11. To compare our system to the related RDF caching techniques we choose as baseline the AET-based caching [39] which caches normalized join execution trees and is the most efficient among the related work systems. We also compare the effectiveness of our techniques for: 1) utilizing more general results using the abstract cache requests, 2) generating *profitable* results using our benefit estimations. Figure 11, shows that our fully functional cache implementation can achieve the best cost saving ratio stabilising to 96% after 350 queries. We also note that the cache efficiency is reduced by nearly 15% when removing our abstract request functionality and another 4% when removing the discovery of *profitable* results. Lastly, our system outperforms the AET based caching by nearly 24% due to its ability to utilize all possibly usable cached results. The cost savings achieved by the AET based technique are limited to the caching of non-selective queries that do not vary much across the workload. The efficiency difference compared to our cache would increase for workloads with more variable query types and filtering values.

7.5 Dataset Size and Caching Policy effects

The left part of Figure 12 depicts the efficiency of our cache for W4, utilizing LUBM10k and LUBM20k. The major differences between the two are:

• The average response time in the beginning of the execution is larger for LUBM20k. When increasing the dataset size, the response time for non-selective queries increases respectively while the performance of selective queries remains almost stable. This is why the baseline average response time increases but remains less than double.

• The time needed for our caching framework to give millisecond-range average response times increases. While for LUBM10k our framework requires 1300 sec, 2000 sec are required for fully caching the general workload for LUBM20k. This is due to the fact that most of the *profitable* queries are non-selective, thus their execution time increases along with the dataset size.

To evaluate the effectiveness of the described caching policy heuristics described in Section 5, we execute W4 three times: 1) with unlimited cache and without using our benefit estimation to discover *profitable* queries, 2) with unlimited cache and using the proposed benefit estimation, 3) with benefit estimation and 2GB cache size. The experimental results are presented on the right part of Figure 12. For the

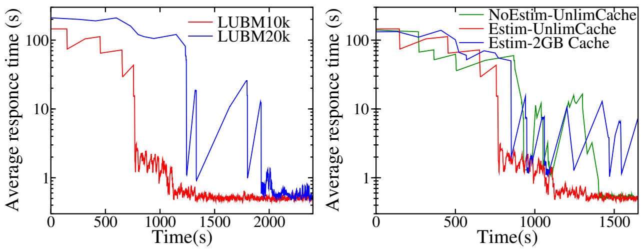

Figure 12: Cache performance for different dataset, caching policies and cache size

no estimation policy, instead of maintaining benefit estimations we just maintain a counter for each cache request, incremented by 1 each time the query graph is requested. We observe that both policies can eventually minimize the average response time of the workload but the proposed simple benefit-based approach manages to smoothly decrease the response times by caching the *profitable* queries in the most suitable order. In contrast, the policy that uses no benefit estimations cannot effectively order the caching of *profitable* results, leading to large deviations in the average response time due to the fact that costly queries happen to get cached later than selective queries.

When limiting the cache size, we note that the large output of LQ15 cannot be cached, while all the rest cached results fit inside our cache, requiring nearly 1.7GB of disk space. LQ15 requires 182 sec to be executed by H_2RDF+. While all other queries present interactive response times after 800 sec, the execution of LQ15 triggers the oscillations of the average response time.

8. CONCLUSIONS

In this paper we presented a novel SPARQL caching framework that is able to effectively cache and utilize query results. We introduced a SPARQL canonical labelling algorithm that manages to generate canonical labels for most real life SPARQL queries in less than a millisecond. Furthermore, we extended the *DPccp* dynamic programming planner by adding support for subgraph cache checks and generation of optimal query plans that consider the utilization of cached query subgraphs. We introduced a Cache Controller module that is able to discover and cache *profitable* SPARQL queries to reduce the response times for several workloads. Our caching framework was integrated on top of a state-of-the-art distributed RDF datastore, reducing its average response time by up to two orders of magnitude and offering interactive response times for complex workloads and huge RDF datasets.

9. ACKNOWLEDGMENTS

Nikolaos Papailiou has received funding from IKY fellowships of excellence for postgraduate studies in Greece - SIEMENS program. This work has received funding from the European Union, Seventh Framework Programme (FP7/2007-2013) under grant agreement no 619706.

10. REFERENCES

[1] Bliss. http://www.tcs.tkk.fi/Software/bliss/index.html.
[2] data.gov.uk. http://data.gov.uk/.
[3] LUBM queries. http://swat.cse.lehigh.edu/projects/lubm.
[4] Nauty and Traces. http://pallini.di.uniroma1.it/.
[5] RDFS. http://www.w3.org/TR/rdf-schema/.
[6] Resource Description Framework (RDF). http://www.w3.org/RDF/.

[7] Saucy. http://vlsicad.eecs.umich.edu/BK/SAUCY/.
[8] SPARQL. http://www.w3.org/TR/rdf-sparql-query/.
[9] Sports Refresh: Dynamic Semantic Publishing. http://www.bbc.co.uk/blogs/bbcinternet/2012/04/sports_dynamic_semantic.html.
[10] V. Arvind and J. Köbler. Graph isomorphism is low for zpp (np) and other lowness results. In *STACS 2000*. Springer, 2000.
[11] M. Atre, J. Srinivasan, and J. Hendler. BitMat: A Main-memory Bit Matrix of RDF Triples for Conjunctive Triple Pattern Queries. In *ISWC*, 2008.
[12] V. Bonstrom, A. Hinze, and H. Schweppe. Storing rdf as a graph. In *Web Congress, 2003*.
[13] P. Gassner, G. M. Lohman, K. B. Schiefer, and Y. Wang. Query optimization in the IBM DB2 family. *IEEE Data Eng. Bull.*, 16(4), 1993.
[14] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2), 2005.
[15] S. G. Hartke and A. Radcliffe. Mckay's canonical graph labeling algorithm. *Communicating mathematics*, 479, 2009.
[16] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum. YAGO2: Exploring and Querying World Knowledge in Time, Space, Context, and Many Languages. WWW, 2011.
[17] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11), 2011.
[18] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *CIDR*, 2011.
[19] J. Köbler, U. Schöning, and J. Torán. *The graph isomorphism problem: its structural complexity.* 1994.
[20] Y. Kotidis and N. Roussopoulos. Dynamat: a dynamic view management system for data warehouses. In *ACM SIGMOD Record*, 1999.
[21] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for sparql. In *ICDE*. IEEE, 2012.
[22] J. Lorey and F. Naumann. Caching and prefetching strategies for sparql queries. In *ESWC*. 2013.
[23] M. Martin, J. Unbehauen, and S. Auer. Improving the performance of semantic web applications with SPARQL query caching. In *The Semantic Web: Research and Applications*. 2010.
[24] B. D. McKay. *Practical Graph Isomorphism.* Department of Computer Science, Vanderbilt University, 1981.
[25] B. D. McKay and A. Piperno. Nauty and Traces User's Guide.
[26] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *ACM SIGMOD Record*, 2001.
[27] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *VLDB*, 2006.
[28] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *SIGMOD*. ACM, 2008.
[29] T. Neumann and G. Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *VLDBJ*, 19(1), 2010.
[30] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2RDF+: High-performance Distributed Joins over Large-scale RDF Graphs. In *IEEE Big Data*, 2013.
[31] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. *H_2RDF*: Adaptive Query Processing on RDF Data in the Cloud. In *WWW*, 2012.
[32] N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, and N. Koziris. H2RDF+: An Efficient Data Management System for Big RDF Graphs. ACM SIGMOD, 2014.
[33] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. *ACM SIGMOD Record*, 2000.
[34] H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, and J. Broekstra. Index structures and algorithms for querying distributed rdf repositories. In *WWW*, 2004.
[35] T. Tran and G. Ladwig. Structure index for rdf data. In *SemData*, 2010.
[36] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. Boncz. Heuristics-based query optimisation for SPARQL. In *ICDT*. ACM, 2012.
[37] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *PVLDB*, 1(1), 2008.
[38] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, 2004.
[39] M. Yang and G. Wu. Caching intermediate result of sparql queries. In *WWW 2011.*
[40] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB*, 6(4), 2013.
[41] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree+ delta<= graph. In *VLDB*, 2007.

APPENDIX

A. WORKLOAD QUERIES

For completeness, we include the LUBM SPARQL queries described in Section 7.3.2.

A.1 Selective query workload (W1):

LQ1: ?x rdf:type ub:GraduateStudent .
?x ub:takesCourse <GraduateCourse0>
LQ3: ?x rdf:type ub:Publication .
?x ub:publicationAuthor <AssistantProfessor0>
LQ4: ?x ub:worksFor <Department0.University0.edu> .
?x ub:name ?n . ?x ub:emailAddress ?em .
?x ub:telephone ?t . ?x rdf:type ub:Professor
LQ5: ?x rdf:type ub:GraduateStudent .
?x ub:memberOf <Department0.University0.edu>
LQ7: ?x rdf:type ub:Student . ?y rdf:type ub:Course .
?x ub:takesCourse ?y .
<AssociateProfessor0> ub:teacherOf ?y
LQ8: ?x rdf:type ub:Student . ?y rdf:type ub:Department .
?x ub:memberOf ?y . ?x ub:emailAddress ?em .
?y ub:subOrganizationOf <University0> .

A.2 Non selective query workload (W2):

LQ2: ?z rdf:type ub:Department .
?x ub:memberOf ?z . ?x rdf:type ub:GraduateStudent .
?z ub:subOrganizationOf ?y . ?y rdf:type ub:University .
?x ub:undergraduateDegreeFrom ?y .
LQ9: ?x rdf:type ub:Student . ?y rdf:type ub:Professor .
?z rdf:type ub:Course . ?x ub:advisor ?y .
?y ub:teacherOf ?z . ?x ub:takesCourse ?z .
LQ15:?p rdf:type ?tp . ?p ub:worksFor ?d .
?s ub:takesCourse ?c . ?p ub:teacherOf ?c

B. EXTENDED EXPERIMENTS

In order to test our system with real-life RDF data and SPARQL queries that contain more edges and selective nodes, we utilize the Yago2 dataset [16]. This dataset consists of real data gathered from various resources such as Wikipedia, WordNet, GeoNames, etc., and contains more than 120 million triples. It is smaller than the various LUBM datasets we used in our experimental section but introduces more complex query and data structures. To test our system, we generate a workload consisting of SPARQL queries, also used in [29], containing up to 10 triple patterns:

YQ1: ?gn <hasGivenName> ?p. ?fn <hasFamilyName> ?p. ?p <type> "scientist?. ?p <wasBornIn> ?city. ?p <hasAcademicAdvisor> ?a. ?a <wasBornIn> ?city2. ?city <isLocatedIn> "Switzerland". ?city2 <isLocatedIn> "Germany".

YQ2: ?a <type> "actor". ?a <livesIn> ?city . ?a <actedIn> ?m1 . ?a <directed> ?m2 . ?city <isLocatedIn> ?s. ?s <isLocatedIn> "United States". ?m1 <type> "movie" . ?m1 <isLocatedIn> "Germany". ?m2 <type> "movie" . ?m2 <isLocatedIn> "Canada".

YQ3: ?p1 <hasGivenName> ?n1 . ?p1 <wasBornIn> "Berlin". ?p1 <isMarriedTo> ?p2. ?p2 <hasGivenName> ?n2 . ?p2 <wasBornIn> "London".

In order to make this workload more challenging, our workload generator utilizes random city and country names. This workload resembles the selective W1 used in Section

7.3.2. All the above queries have complex query graph structures with several triple patterns but their selectivity is small and thus the average execution time of the workload without caching is in the range of 2.5 seconds.

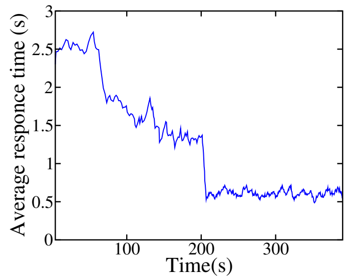

Figure 13: Yago2 query workload

As depicted in Figure 13, our caching framework can gradually reduce the average query execution time for this workload after nearly 200 seconds. This time is required to execute and index the 3 abstract queries that contain information for all possible selective IDs. In fact, query structure complexity (number of triple patterns, graph structure) mainly affects the planning time of the query, as was tested in Section 7.2, while the execution complexity of the query mainly depends on the underlying data and join algorithms. Therefore, the major conclusion of this experiment is that our system has the same effects on workloads with complex query structures. Its overhead depends on the planning overhead, which is in the range of milliseconds, while the join execution complexity occupies the major part of the query time, especially in the case of large non-selective queries. As depicted in this experiment, even for small selective queries the query structure complexity introduces negligible overheads to the average query response times.

C. ALGORITHMS

In this section, we formally describe our proposed algorithms using pseudocodes. We start with the *checkCache* method described in Algorithm 1. This algorithm is used inside the *DPccp* planner to find usable cached results for every connected subgraph (csg) of the abstract query graph. The abstraction procedure, as mentioned in Section 4.1, removes all the bound subject and object nodes and introduces the respective filters that are contained in the auxiliary query info (aux). This auxiliary query info also contains the filtering, projection, grouping and result ordering information of the initial query. Furthermore, the *checkCache* method also searches for the k most relevant results in the ResultCache. The major steps of this algorithm are the canonical labelling and the top-k tree search on top of the respective result tree. An important part of the *searchTree* method is the selectivity estimation of a cached result edge according to the auxiliary query info, described using function *selectivity*$(edge, aux)$. This function computes: 1) the usability of cached results for the query in hand, 2) the selectivity of the usable cached results using the abstract query estimator (line 42) of Section 4.1.

Another important algorithm of our caching framework is the *cacheResult* Algorithm 2 that inserts the meta-data of a result in our cache structure. Again, the algorithm consists of two major steps, the canonical labelling and the tree insertion to the respective result tree. The *treeInsert* method, as mentioned in Section 4.1, is responsible for creating the respective tree node along with the parent nodes that do not already exist. It also updates the *minResults* value of each node that takes part in the insertion procedure by propagating the result size from the leaf node to the root node, applying selectivity estimations when crossing indexed edges.

Our next algorithm is the *addBenefit* Algorithm 3 that attributes the benefit for a specific query subgraph using

Algorithm 1 CHECKCACHE

```
1: function checkCache($V, E, aux, k$)
2:     //$V, E$ : vertices and edges of the abstract query subgraph
3:     //$aux$ : auxiliary query info(filters, projections, etc)
4:     //$k$ : search for the top-k results
5:     $label \leftarrow canonicalLabel(V, E)$
6:     $resultTree \leftarrow ResultCache.get(label)$
7:     //$resultTree$ : the result tree for a certain canonical label
8:     return $searchTree(resultTree, aux, k)$
9: function $searchTree(resultTree, aux, k)$
10:     $results \leftarrow \{\}$
11:     //$openNodes$ : priority queue with pairs (node, cost)
12:     $openNodes \leftarrow \{(resultTree.root, 1)\}$
13:     while $openNodes \neq \{\}$ do
14:         //get the open node with the minimum cost
15:         $n \leftarrow openNodes.removeHead()$
16:         if $(results.size = k)$and$(n.cost \geq results.maxCost)$ then
17:             //We have found k results and all open nodes
18:             //have greater cost than the current results
19:             return $results$
20:         $processNode(n, openNodes, results, aux, k)$
21:     return $results$
22: function $processNode(n, openNodes, results, aux, k)$
23:     for $(edge, child) \in n.children()$ do
24:         if $(s = selectivity(edge, aux)) > 0$ then
25:             $child.selectivity \leftarrow s * n.selectivity$
26:             $child.cost \leftarrow child.minResults * child.selectivity$
27:             if $child.isLeaf()$ then
28:                 //maintain the k best results
29:                 if $results.size < k$ then
30:                     $results.add(child)$
31:                 else if $results.maxCost > child.cost$ then
32:                     $results.removeMaxAndAdd(child)$
33:             else
34:                 $openNodes.add(\{child, child.cost\})$
35: function $selectivity(edge, aux)$
36:     //compute edge selectivity for $aux$ info
37:     $selectivity \leftarrow 1$
38:     for $a \in aux.get(edge.variable)$ do
39:         //check edge usability
40:         if $edge.subsumes(a)$ then
41:             if $a.isFilter()$ then
42:                 if $edge.isIndexed$ then
43:                     //filter selectivity estimation
44:                     //using the abstract result estimator
45:                     $selectivity \leftarrow selectivity * filterSelectivity(a)$
46:             else if $a.isJoinVariable()$ then
47:                 if $edge.isNotIndexed()$ then
48:                     $selectivity \leftarrow selectivity * 2$
49:         else
50:             $selectivity \leftarrow 0$
51:     return $selectivity$
```

its canonical label. This algorithm, introduced in Section 5.1, follows all possibly usable results in the benefit tree (at least 3 for each tree level, see lines 20-22) and attributes the respective benefit, computed using its selectivity estimation and Equation 2. It also prunes sub-trees with benefit $\leq$ 0 and adds the query ID to the list of the query IDs that contributed to the benefit of a cache request.

Algorithm 4 describes the complete processing of a SPARQL query. It starts with the abstraction of the query graph updating its *aux* info. It then calls the extended *DPccp* planner that issues cache requests and records their benefit using Equation 1. The optimal query plan is executed and then the CacheController thread handles the caching of produced intermediate results, the request and result benefit attribution (lines 8-10). The *cacheResults* method tries to cache all computed results consulting our caching policy. The *addRequestBenefits* updates benefits for all recorded cache requests using Algorithm 3. Lastly, the *addResultBenefit* method updates the benefit of all utilized cached results. It computes their contribution to the query execution by checking the optimal execution time computed by the *DPccp* without the use of the respective cached result.

Our caching policy, described in Section 5.2, is depicted in Algorithm 6. The *evictions* function, called if the cache constraints are violated, tries to find a set of results that cover the new result size and have cumulative benefit that is lower than the benefit of the new result. Furthermore, the *decreaseBenefits* function is presented that handles the update of benefits after the execution of a profitable result mentioned in the end of Section 5.1.

Lastly, Algorithm 5 presents our periodic processes. The *updateBenefits* method is responsible for decreasing the benefit estimations for both cached results and requests through time. To do so it uses a configurable decrease factor $0 < a < 1$. It also maintains the ordered lists of results and requests used in our other algorithms. In the case of cache requests, we remove requests that are not in the top-k most profitable ones and also compute their execution cost. The *profitableQueryGeneration* method iterates over all cache requests in decreasing order of benefit/cost and, consulting our caching policy, selects the most profitable request for execution. It proactively checks the caching policy using an estimation of the query size, in order to avoid executing requests that cannot be cached due to their size and benefit constraints.

Algorithm 2 cacheResult

```
function $cacheResult(V, E, aux, size)$
    //$V, E$ : vertices and edges of the abstract query graph
    //$aux$ : auxiliary query info(filters, projections, etc)
    //$size$ : the size of the result in records
    $label \leftarrow canonicalLabel(V, E)$
    $resultTree \leftarrow ResultCache.get(label)$
    $treeInsert(resultTree.root, aux, size)$
function $treeInsert(node, aux, size)$
    if $node.isLeaf()$ then
        $node.minResults = size$
    else
        $aInfo \leftarrow aux.get(node.variable)$
        if $((edge, child) = node.getEdge(aInfo)) = null)$ then
            //Edge does not exist, create new
            $(edge, child) = newEdge(aInfo)$
        $treeInsert(child, aux, size)$
        if $aInfo.isIndexed()$ then
            //compute the maximum selectivity of the index
            $maxSel \leftarrow maxSelectivity(edge)$
            $results = child.minResults * maxSel$
        else
            $results = child.minResults$
        if $results < node.minResults$ then
            if $results \leq 1$ then
                $results \leftarrow 1$
            $node.minResults = results$
```

Algorithm 3 addBenefit

```
function $addBenefit(label, aux, benefit, qID)$
    //$label$ : the canonical label of the abstract query graph
    //$aux$ : auxiliary query info(filters, projections, etc)
    //$benefit$ : the estimated benefit for the query
    //$qID$ : the query ID
    $benefitTree \leftarrow CacheRequests.get(label)$
    $treeAddBenefit(benefitTree.root, aux, benefit, 1, qID)$
function $treeAddBenefit(node, aux, benefit, s, qID)$
    //$s$ : parent node selectivity
    if $node.isLeaf()$ then
        $b = benefit - s * R/thr$ //Equation 2
        if $b > 0$ then
            $node.benefit = node.benefit + b$
            $node.queryIDs.add(qID)$
    else
        $aInfo \leftarrow aux.get(node.variable)$
        $newEdgeIfNotExists(“*”)$
        $newEdgeIfNotExists(“*, Index”)$
        $newEdgeIfNotExists(aInfo)$
        for $(edge, child) \in node.children()$ do
            //check usability, selectivity of existing edges
            $selectivity \leftarrow s * selectivity(edge, aInfo)$
            //prune subtrees with $benefit \leq 0$
            if $(benefit - selectivity * R/thr) > 0$ then
                $treeAddBenefit(child, aux, benefit, selectivity)$
```

Algorithm 4 executeQuery

```
function $executeQuery(query)$
    $(q, aux) \leftarrow abstractQuery(query)$
    //q: abstract query, aux: auxiliary query info
    $cacheRequests \leftarrow \{\}$
    $plan \leftarrow DPccp(q, aux, cacheRequests)$
    $results \leftarrow execute(plan)$ //RDF engine
    //handled offline by the CacheController thread
    $CacheController.cacheResults(results)$
    $CacheController.addRequestBenefits(cacheRequests, qID)$
    $CacheController.addResultBenefit(q, plan)$
function $cacheResults(results)$
    //cache computed results
    for $result \in results$ do
        //get the respective benefit from the cache requests
        $request \leftarrow CacheRequests.get(result)$
        $result.benefit \leftarrow request.benefit$
        $result.queryIDs \leftarrow request.queryIDs$
        $cache(result, result.benefit)$
function $addRequestBenefits(cacheRequests, qID)$
    for $(req, benefit) \in cacheRequests$ do
        $addBenefit(req.label, req.aux, benefit, qID)$
function $addResultBenefit(q, plan)$
    //update the benefit of utilized cached results
    for $result \in plan.usedCachedResults$ do
        $newPlan \leftarrow DPccpWithoutResult(q, result)$
        $result.benefit+ = (newPlan.cost - plan.cost)$
```

D. COMPLEXITY

In this section, we formally examine the time and space complexities of our proposed algorithms. Our first algorithm is the canonical labelling algorithm proposed in Section 3. The complexity of this algorithm is directly associated with the complexity of the *Bliss* algorithm that attempts to solve the graph isomorphism (GI) problem. GI is known to have time complexity at most $O(2^{\sqrt{n \log n}})$ for graphs with n vertices [10]. However, this is not a representative bound for *Bliss* because its complexity mainly depends on the amount of automorphisms present in the graph structure rather than on the number of its vertices. Indeed, there are graph examples that result in exponential labelling times but for general graphs *Bliss* presents sub-exponential complexity. Our extended SPARQL query labelling algorithm introduces a polynomial time, $O(n)$, transformation of the input query which is negligible compared to the worst-case exponential time complexity of *Bliss*. The major overhead of our scheme is the fact that it transforms the n vertices of the query graph to $3n$ vertices and thus introduces a polynomial increase to the input size. However, this overhead does not substantially affect the worst-case complexity of *Bliss*.

Our second algorithm is the top-k, A* search, (searchTree in Algorithm 1) performed in order to search results that share the same canonical label. The irregularity of the A* search does not allow us to set a useful upper bound for this algorithm. Of course, an upper bound for the algorithm is the maximum size of the tree stored inside each cache record but this bound does not take into account the intelligent pruning of the search space performed. In the worst case, the tree search will have complexity O(r), where r is the maximum amount of cached results that share the same graph structure and thus the same canonical label. We note here that r is also bound by the cache size constraints and we can therefore expect that it will not grow limitless.

Our checkCache Algorithm 1, generates a canonical label and then performs a tree search in the respective result tree. As mentioned before, the canonical labelling time will, in the worst case, dominate the checkCache mechanism due to its

Algorithm 5 Cache Controller Periodic Process

```
// methods that run in configurable time or query intervals
function $updateBenefits$
    $//OrderedResults$ : benefit ordered list of cached results
    for $result \in ResultCache$ do
        //decrease benefit with time using paramater $0 < a < 1$
        $result.benefit = result.benefit * a$
        $OrderedResults.insert(result)$
    $//OrderedRequests$ : benefit ordered list with max size
    for $request \in CacheRequests$ do
        //decrease benefit with time
        $request.benefit = request.benefit * a$
        $OrderedRequests.insert(request)$
    //remove cache requests that are not in OrderedRequests
    $removeFromCacheRequests(OrderedRequests)$
    for $request \in OrderedRequests$ do
        //estimate cost for best requests using $DPccp$
        $request.benefit = request.benefit/estimateCost()$
function $profitableQueryGeneration$
    //iterate in decreasing order of benefit/cost
    for $req \in OrderedRequests$ do
        //proactively check cache replacement
        $evict \leftarrow evictions(estimateSize(req), req.benefit)$
        if $evict.satisfied$ then
            $result \leftarrow executeQuery(req)$
            return
```

Algorithm 6 cachingPolicy

```
function $cache$(result, benefit)
    $evict \leftarrow evictions(result.size, benefit))$
    if $evict.satisfied$ then
        $removeCachedResults(evict)$
        $cacheResult(result.V, result.E, result.aux, result.size)$
        $decreaseBenefits(result.queryIDs)$
        return
function $evictions(size, benefit)$
    //cache replacement policy
    $evict \leftarrow \{\}$
    if $availableCacheSize \geq size$ then
        $evict.satisfied = true$
        return $evict$
    //iterate cached results in decreasing benefit order
    for $result \in OrderedResults$ do
        if $result.benefit \leq benefit$ then
            $evict.add(result)$
            if $evict.totalSize \geq size$ then
                break
    if $evict.totalSize \geq size$ then
        $evict.satisfied = true$
    else
        $evict.satisfied = false$
    return $evict$
function $decreaseBenefits(queryIDs)$
    for $request \in CacheRequests$ do
        $oldSize \leftarrow request.queryIDs.size$
        //remove common query IDs
        $request.queryIDs = request.queryIDs \setminus queryIDs$
        $newSize \leftarrow request.queryIDs.size$
        $request.benefit = request.benefit * newSize/oldSize$
```

exponential time complexity and due to the fact that the tree search has bounded worst-case performance. However, as shown in our experimental evaluation, our checkCache mechanism is practical and presents acceptable time complexity for general graphs.

We now examine the time complexity of our dynamic programming query planner that integrates the *DPccp* algorithm with our checkCache algorithm. *DPccp* bases its enumeration procedure on finding all csg-cmp-pairs in the query graph [28]. Csg-cmp-pairs are pairs containing a connected subgraph(csg) of the query graph and one of its connected complement subgraphs(cmp).

Definition 4. *(csg-cmp-pair). Let G=(V,E) be a query graph and S1, S2 two subsets of V such that S1 $\subseteq$ V and S2 $\subseteq$ (V $\setminus$ S1) are a connected subgraph and a connected complement respectively. If there further exists an edge (u, v)$\in$E such that u$\subseteq$S1 and v$\subseteq$S2, we call (S1, S2) a csg-cmp-pair.*

For each unique csg, connected subgraph of the query graph, we call the checkCache algorithm in order to search for usable cached results. Therefore, if s is the amount of connected query subgraphs and m is the amount of csg-cmp-pairs, the complexity of our planner is $O(s \cdot c + m)$ where c is the complexity of the checkCache mechanism. The complexity of the original *DPccp* is $O(m)$. In the worst case m is exponentially larger than s because for each csg we need to enumerate all the respective cmp subgraphs. In fact, for the worst case of clique graphs $s = O(2^n)$ and $m = O(3^n)$ [27]. Therefore, the worst-case complexity of our extended dynamic programming planner is $O(2^n \cdot 2^{\sqrt{n \log n}} + 3^n)$ which does not largely deviate from the original complexity of the *DPccp* algorithm. However, as depicted in our experiments and in the extensive complexity evaluations of both *DPccp* [27] and *Bliss* [1], the above worst-case complexity does not arise for general graphs and the algorithms are practical for all tested query graphs.

We now discuss the complexity of our offline operations handled by the CacheController module. By offline we mean that they are not part of the critical path of executing a SPARQL query. Initially, we study the complexity of the cacheResult method presented in Algorithm 2. This operation consists of a canonical labelling step and one execution of the treeInsert method. The complexity of the treeInsert method is linear in the amount of variables in the query graph and thus is linear, $O(n)$, in the input graph size. The treeInsert method has also linear space complexity because it only inserts one node in the result tree along with its respective parent nodes that do not already exist. Therefore, the complexity of the cacheResult method is dominated by the worst-case complexity of our canonical labelling.

The *addRequestBenefits* method, presented in Algorithm 4, is responsible for attributing benefits to all possibly usable query patterns for all the cache requests issued during the planning of a query. The number of cache requests for a query graph is bound by the number of its connected subgraphs(csg) s. For every csg the addBenefit method is called (Algorithm 3). The worst-case time and space complexity of the addBenefit method is $O(d^v)$ where d is the maximum degree of the tree nodes (at least 3) and v is the number of variables in the query. In total, the worst-case complexity of the *cacheRequests* method is $O(s \cdot d^v)$. This worst-case complexity is tamed by: (i) the offline pruning of the benefit trees that maintains the top-k most profitable tree nodes for all query labels and (ii) the ability of the addBenefit method to prune entire subtrees with benefit ≤ 0.

The *profitableQueryGeneration* method, presented in Algorithm 5, iterates, in a benefit order, over the benefit estimations checking the caching policy in order to select the query that is going to be executed and cached. Our caching policy check has linear complexity, $O(res)$, in the amount of cached results res because, in the worst case, it needs to check the removal of all cached queries. Therefore, its worst-case complexity is $O(req \cdot res)$ where req is the number of maintained request benefit estimations. As stated above, this number is regulated by our offline request pruning process in order not to grow exponcentially. The res parameter is also regulated by the cache size constraints.

Our *updateBenefits* method, presented in Algorithm 5, has complexity $O(res \cdot \log res + req \cdot creq + req \cdot \log req + req \cdot dp)$. It sorts res cached results, it maintains and sorts req of the $creq$ accumulated requests. It also executes req times the *DPccp* algorithm, with complexity dp as presented above. The main memory space complexity of our caching framework is $O(res + req + creq)$ which is regulated using the configurable res and req values. The $creq$ value is regulated by the execution frequency of the *updateBenefits* method.