A Provenance-Based Caching System to Speed-up SPARQL Query Answering

Gianmaria Silvello[1,*], Dennis Dosso[2]

[1]University of Padua, Via Gradenigo 6/b, Padova, 35131, Italy
[2]Siav S.p.A., Italy

Abstract
We propose a new provenance-based approach for the dynamic extraction of subgraphs from an RDF database that can be used as a cache for a faster response to SPARQL queries. The proposed system can deal with *dynamic workloads*, allowing us to use the cache to answer queries similar but not identical to others seen before by the system.

Moreover, our approach exploits existing technologies and can be seamlessly integrated on top of existing RDF DBMS without requiring any change to their internals. We conducted an in-depth evaluation on a widely used synthetic database (BSBM) and on Wikidata using real-world query logs. We show that the proposed approach improves the query execution times over the whole database.

Keywords
SPARQL, Caching, Provenance, RDF datasets

1. Introduction

In recent years, the Web of Data has continuously expanded, accompanied by a new generation of semantically enhanced applications that leverage this infrastructure to develop novel services [1]. In this context, the Resource Description Framework (RDF) and the SPARQL query language are the cornerstones to represent, share, and query resources [2]. RDF can represent, query, and exchange heterogeneous data in distributed and unconstrained environments [3, 4]. These technologies play a central role in the creation and access of *Knowledge Graphs* (KG), which have gained increasing prominence in both academic and industrial settings [5]. However, RDF introduces challenges for downstream applications, especially when efficiently querying large graphs in real time is needed.

Being able to query large KGs in real-time efficiently is a key to improving their use in downstream applications such as (1) exploratory search and (2) integration with *Large Language Models* (LLMs). Exploratory search on KGs arises when a user needs to understand and extract insights from an unfamiliar KG. In these exploratory sessions, the users issue a series of SPARQL queries to identify relevant portions of the KG that can answer their questions [6]. Whereas integrating LLMs with the symbolic knowledge encoded in KGs is a promising avenue for enhancing the precision of factual query answering by LLMs [7].

SEBD 2024: 32nd Symposium on Advanced Database Systems, June 23-26, 2024 - Villasimius, Sardinia, Italy
gianmaria.silvello@unipd.it (G. Silvello); dennis.dosso@siav.it (D. Dosso)
https://www.dei.unipd.it/~silvello/ (G. Silvello)
0000-0003-4970-4554 (G. Silvello); 0000-0001-7307-4607 (D. Dosso)

These applications are highly interactive and/or resource-intensive and, in both cases, require fast answering times where query completeness can be a tradeoff with query efficiency, especially in the first phases of KG exploration and integration [8]. There is vast work in the literature [2, 9, 10, 11] to improve the performances of RDF DBMS in answering SPARQL queries. Two main directions have been pursued, focusing on *static workloads* – i.e., fixed sets of known queries not changing over time – and on *dynamic workloads* – i.e., unknown queries that may have little or no similarity with queries issued before – that better suit RDF-based real-world Web applications. Working with dynamic workloads is generally hard since caching already seen queries only partially covers the workload and the results may be very big. Moreover, understanding if two queries are equivalent or similar (pairwise similarity) is a difficult problem [12]. Query containment for SPARQL fragments is NP-complete for conjunctive queries and undecidable for general queries [13, 14, 15]. Thus, it is also not practical to face the caching task by resolving to check for each new query if it is contained in some previously answered query [15].

Despite these difficulties, the *dynamic workload* research direction is well-suited to target the typical usage scenario of a SPARQL endpoint represented by an agent who *"queries the data and gradually refines the search until the desired result is obtained"* [16]. The users interested in RDF data typically use programmatic query clients to retrieve information from SPARQL endpoints [17] producing subsequent queries with similar query patterns differing in specific attributes of triple patterns. Zhang et al. [18] also noticed that many queries issued to SPARQL endpoints are robotic queries, i.e., queries produced by automated scripts or programs for index size inferring, data crawling, or malicious attacking. As an example of this phenomenon, in the query log datasets explored in [18], 90% of queries are provided by only 0.4% automated software programs and, of these, the percentage of unique query templates was 0.28%. Further evidence is given by Bonifati et al. [16] who analyzed a large query log from DBpedia highlighting a high number of unique queries with long *streaks* of similar queries repeated in narrow timeframes that call for *"optimization techniques for sequences of similar queries"*. This aspect is further emphasized if we consider that the query load and server performance of another widely-used KG as Wikidata are dominated by robotic queries [19] and that such queries report a higher cluster similarity than organic ones [20].

Following these observations, we propose a new workload-adaptive approach based on data provenance [21] that allows us to detect the triples in an RDF graph responsible for the computation of a query result set. We propose a system – the Cache-Based Architecture (CBA) – employing a cache to store the triples in the lineage of past-seen queries and leverage them to quickly answer similar, but not necessarily identical, queries without adding any computational complexity to the standard query processing.

We evaluated CBA on BSBM, a synthetic database, instantiated to 250K, 1M, 25M, and 100M triples to check system scalability and on Wikidata using a real query log. The evaluation shows that the query provenance can be used to answer previously unseen queries, presents sizeable speed-ups achievable with different cache sizes, shows that the system scales well to large databases and query logs, and achieves good completeness even for first-time queries.

The advantages of CBA are that (i) it natively exploits the technologies already implemented by RDF DBMS without modifying them or developing a new system from scratch. As such, it can be easily deployed on top of already functioning systems; (ii) it can be used as a system to create smaller databases composed of the most relevant triples used by the queries coming

```
Q1: SELECT DISTINCT ?p ?l ?price
WHERE {
?p label ?l .
?p type product_type_1 .
?p price ?price .
?p producer producer_1 .
FILTER (?price < 200) }
```

Q1

```
Q2: CONSTRUCT { ?p label ?l .
?p type product_type_1 .
?p price ?price .
?p producer producer_1 . }
WHERE { ?p label ?l .
?p type product_type_1 .
?p price ?price .
?p producer producer_1 .
FILTER (?price < 200) }
```

Q2

```
Q3: SELECT DISTINCT ?P ?PF1 ?PF2
WHERE { ?product rdfs:label ?label .
?product a ?P .
?product bsbm:productFeature ?PF1 .
?product bsbm:productFeature ?PF2 .
?product bsbm:productPropertyNumeric1 ?value1 .
FILTER (?value1 > 300)
FILTER (?PF1 != ?PF2)
} ORDER by ?label
LIMIT 1000
```

Q3

Figure 1: SPARQL queries: Q1 is used by the running example; Q2 is used to calculate the lineage of Q1; Q3 is an example of a BSBM query.

from large dynamic workloads; and (iii) it helps users in scenarios where they need (even partial) results quickly to refine their query, for instance, in the case of tasks such as exploratory search [22] and exploratory analytics [23].

The rest of the paper is organized as follows: Section 2 presents some preliminaries RDF, SPARQL, and data provenance) and introduces the running example; Section 3 illustrates the CBA system architecture; Section 4 describes how provenance and data impact are calculated, and the cache is maintained and updated; Section 5 reports the experimental setup and reports the results of the evaluation; Section 6 presents some related work; and, Section 7 draws some conclusions and outlines future work.

2. Preliminaries

The RDF is a family of specifications developed and supported by the W3C [1] to represent interlinked data on the Web. The key component of RDF is the *statement* – a `<subject, predicate, object>` triple-based structure [24]. We assume the pairwise disjoint infinite sets I (IRIs), B (blank nodes), and L (literals). Assume also an infinite set of variables X, disjointed from the above sets. A triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called *RDF triple*. An *RDF graph* is a set of RDF triples. A *triple pattern* is an element of the set $TP = (I \cup X) \times (I \cup X) \times (I \cup L \cup X)$. A Basic Graph Pattern (BGP) is a finite set of triple patterns $\{t_1, \dots, t_n\}$.

We focus on BGP queries because they are the most frequent type of queries [25] issued to SPARQL endpoints. More precisely, we consider queries with a basic graph pattern that can also contain the `optional` and `filter` clauses. For brevity, we call them BGP+ queries. A built-in condition is constructed using elements from the set $I \cup L \cup V$, constants, logical connectives, inequality and equality symbols, and other features [24]. We do not consider the queries employing the negation clause (`not exists`). The negation (`not exists`) can also be achieved by combining the `optional` and `filter` clauses, but we exclude this option from the queries considered in this work. Moreover, we exclude the queries employing property paths.

BGP contains the triple patterns to be matched onto the RDF data searching for specific subgraphs in the database. The evaluation of a SPARQL pattern returns a multi-set of solution mappings, M, that match the pattern to the data. A solution mapping is a partial function $\mu : X \to T$, where T is a set of triples.

[1]https://www.w3.org/TR/rdf11-primer

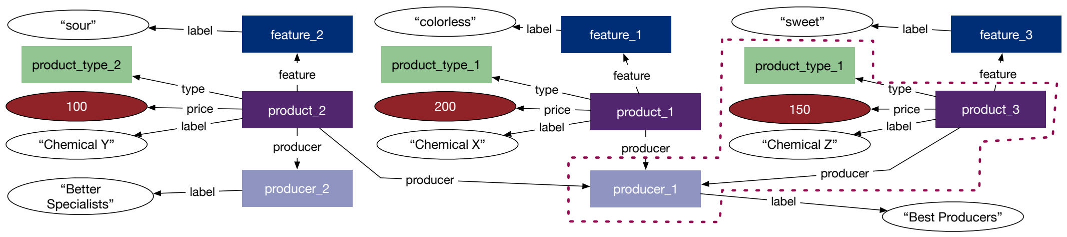

Figure 2: RDF graph representing the running example, inspired by BSBM. It includes entities representing some products with their characteristics. The subgraph within the dotted line is the lineage of query `Q1`.

Figure 2 reports a simple RDF graph representing a use case with products connected to features (e.g., type and price) and producers. The query `Q1` shown in Figure 1 is an example of BGP$^+$ query with the `filter` operator. The result is `<product_3, Chemical Z, 150>`.

Data Provenance is a form of metadata describing the life-cycle and origin of data. It has been widely studied within the database, distributed systems, and Web communities [26, 21]. Works as [27, 28, 29] defined types of data provenance on graph databases along with different algorithms to compute them [30, 31, 32].

In this work, we employ *lineage* because it is one of the most diffused, intuitive, and computationally efficient form of data provenance. For relational databases, the lineage of an output tuple [21, 33] is defined as the set of *all and only* the tuples in the input database that have a role in the generation of that output. Similarly, the lineage of a BGP$^+$ query can be defined as the set of all and only the triples in the input RDF graph that match the query pattern and thus have a role in computing the output. For example, the lineage of query `Q1` is the set of triples highlighted with the dotted line in Figure 2.

In this paper we use `construct` queries as a means to compute lineage since it is a simple and effective method that fits our use cases. Another way to calculate the lineage, more efficient but requiring to change the DBMS internals, is to get the query planner to return the pattern matching along with the result set of the query.

3. System Architecture

The main components of the proposed Cache-Based Architecture (CBA) are reported in Figure 3. We can see that when a query is submitted to the system, a series of operations are carried out online (red arrows) and offline (green arrows). Given a user query, the system first issues it against the cache (step 1), which contains a subgraph of the database. If a cache is hit and an answer can be built, the result is immediately returned to the user. In case of a cache miss, the query is submitted to the main database to get the result.

Another supporting cache called *Empty Result Sets Cache* (ERSC) is employed to keep track of the queries returning the empty set. ERSC is a list of hashed (SHA-256 hashing algorithm) queries stored as strings in a side relational table. The ERSC is particularly useful for databases like Wikidata, where robotic agents sequentially submit identical queries. When these queries present an empty result set, such a support cache reduces the answer times drastically. ERSC is

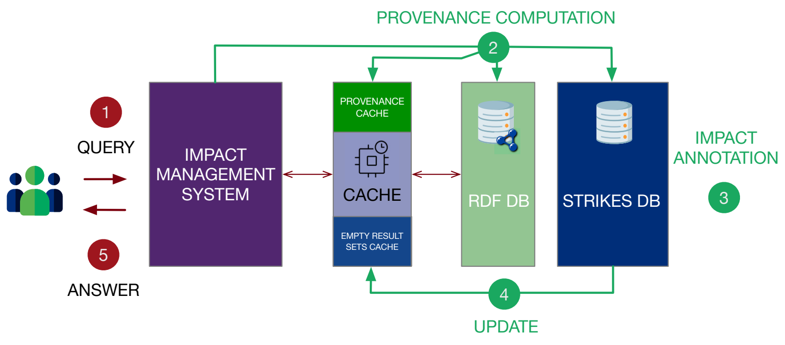

Figure 3: Representation of the main blocks composing the proposed CBA.

highly efficient, requiring minimal disk space since it stores only strings.

In a convenient moment and thus offline, the system calculates the query provenance on the RDF graph (step 2). The query's data provenance provides the complete set of triples involved in generating the output data. In step 3, the triples in the provenance set are used to update a relational database – i.e., the "strike database" – which keeps track of how many times a triple has been used by the workload so far – i.e., the triple number of strikes. The number of strikes is the basis for calculating the *impact* of a triple within the RDF graph. The triples' impact is periodically used to update the cache (step 4), deciding which triple has to be added or removed. The more frequently the system updates the cache, the faster it adapts to a dynamic workload.

Another supporting cache we employ is the "provenance cache" (PC) containing the provenance of the queries. It also becomes beneficial when identical queries are submitted many times over. We maintain the PC size under control by employing a standard Least Recently Used (LRU) algorithm. Note that PC is used offline and can be maintained on disk; thus, it has no impact on the system runtime and a low impact on the system's requirements.

For the supporting relational database, one main table contains the triples of the cache together with the total number of strikes of each triple. This information is required to keep the cache updated and limit its size. A second relational table tracks when the strikes were assigned to implement the refresh algorithms described below.

This architecture provides quick access and a fast computation for several queries. It can be built on top of any database system by employing existing technologies with every DBMS. The proposed architecture does not require modifying the support DBMS or implementing additional indexes or view selection mechanisms. This architecture can also be realized with different memory systems and in a distributed manner. It can be easily integrated with other indexing or caching techniques further to improve the execution time of dynamic query workloads.

4. Methods

Provenance Computation The lineage of BGP$^+$ queries can be computed by issuing a `construct` query on the RDF graph using the same BGP of the corresponding `select` query. In the case of queries including the `optional` operator, triples not originally present in the graph may be used in the `construct` query. In this case, `ask` queries allow us to check if a triple is present in the original database or not and to avoid including it in the lineage. We can see that `Q2` returns all the subgraphs that match the specified pattern present in the `construct` and in the `where` clauses.

Data Impact Let G be an RDF graph and $W = \{Q_1, \ldots, Q_n\}$ a query workload, where $\forall i \in [1, n], \mathscr{L}(Q_i)$ is the lineage of Q_i. Then, the strike count of a triple $t \in G$, is defined as $s(t) = \sum_{i=1}^{n} \mathbb{1}_{\mathscr{L}(Q_i)}(t)$. $\mathbb{1}_{\mathscr{L}(Q_i)}(t)$ is an indicator function that returns 1 if the triple t is in the lineage $\mathscr{L}(Q_i)$ and 0 otherwise. Hence, the total strike count of a triple t is the number of times t has been used by the queries of the workload.

Based on the strike count of a triple, we compute its impact as $k(t) \in \mathbb{R}$ describing the *importance* of a triple in a given context – i.e., the workload in our case. The more a workload uses a triple, the higher its impact. We employ a *log-based impact function* such that, given a triple $t \in G$ with strike count $s(t)$, its impact is computed by the function $k : G \to \mathbb{R}$, $k(t) = ln(1 + s(t))$.

This function guarantees that the triples that were never used have zero impact. The logarithm quickly raises the quantity of impact of a triple. The per-strike increment gets less marked as the strike count increases, i.e., the importance of the triple is already established. The idea is to favor the addition of newly used triples into the cache and keep the growth of frequently used triples contained.

Update phase In the update phase, the cache is updated by deciding which triples should enter or leave it. We define a threshold-based *update strategy* based on the triple impact.

[Threshold-based update strategy] Given an RDF graph G, a triple $t \in G$ with impact $k(t)$ and a threshold value $\tau \in \mathbb{R}_{\geq 0}$, the threshold-based update strategy is $u_\tau : G \to \{0, 1\}$, such that:

$$u_\tau(t) = \begin{cases} 1 & \text{if } k(t) > \tau \\ 0 & \text{if } k(t) \leq \tau \end{cases}$$

When $u_\tau(t) = 1$, a triple is added (or maintained) in the cache, otherwise it is removed. If $\tau = 0$, a triple needs to be used just once by the workload to be added in the cache at the next update. The higher the τ, the higher the triple impact required to enter the cache.

The system runs the update strategy every time an *epoch* is elapsed. An *epoch* is the amount of time or the number of queries after which the system updates the cache, and it is a system parameter. As a rule of thumb, if a database has a high workload, an epoch could be based on time (e.g., one hour or one day); whereas, if the workload is low, an epoch could be based on the number of issued queries to allow the system to gather enough strikes to make the update of the cache worth the effort.

To contain the cache's size and avoid it growing too much, we set an upper bound to the cache size. We define the size as the number of triples stored in the cache, specified by the parameter γ. With this strategy, the cache grows if its overall size is less than or equal to γ. When the cache is full, and we want to insert new triples, we employ a LRU-based strategy, where the least recently used triples are removed from the cache to make space for the new ones. In particular, when the cache size needs to be reduced, the system finds the least recently used set of triples belonging to the same lineage and removes them from the cache. This operation is iterated until the cache size gets below γ. The strike count of the released triples is also set to zero, so to avoid that, they immediately get back in the cache at the next update phase.

5. Experimental Evaluation

All the experiments were run on a Linux Machine with architecture Intel x86_64. The Slurm[2] workload manager was used to run the experiments by assigning to each job one processor and 8GB of RAM. We tested the proposed architecture on two datasets both largely used and reference points for the community: the Berlin SPARQL BenchMark (BSBM)[3] synthetic database and Wikidata[4]. We used a PostgreSQL (v14.1) relational database as a support database. The source code is written in Java, using the rdf4j[5] (for BSBM) and the Virtuoso[6] and the Virtuoso Jena RDF Data Provider[7] (for Wikidata) as API to manage the RDF datasets and run the SPARQL queries. In the case of rdf4j, for each experiment, in each database and its corresponding cache we used a default spoc index. In the case of Virtuoso, the default settings were kept.

Given the controlled environment where we conduct the experimental evaluation, we can define a very efficient system – i.e. the Hash-Based Architecture (HBA) – to determine a reasonable lower bound that can be obtained with a caching system. Indeed, the experiments are carried out with a controlled number of synthetic queries (in the case of BSBM) or robotic ones (in the case of Wikidata).

HBA is based on a hash table containing the hashed queries and their result set. When a query is submitted to HBA, it is hashed and saved in the support database along with its entire result set. To control the size of the table, we put an upper limit to the total number of entries that can be stored, i.e., to the sum of the cardinalities of the result sets stored in the table. We implemented a LRU strategy to refresh the table when it becomes too big.

BSBM is a synthetic database benchmark simulating commercial products and related features. We instantiated the database in different sizes[8]. The queries for BSBM are generated from the query templates proposed in the "Explore Use Case"[9], – i.e., a set of query templates that simulates a consumer looking for products and their features and that are meant to challenge the DBMS. There are twelve query templates available, covering different query patterns. We used the eight templates that can be answered with BGP+ queries (the sole modification regards query pattern three where we removed the negation clauses), each representing a different information need. The templates differ in the structure of their query pattern, i.e., the number of triples they contain, the type of `filter` operators used, and the selection of variables required. These queries can be considered "hard" since they are thought to stress the query engine.

[2]https://slurm.schedmd.com/
[3]http://wbsg.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/BenchmarkRules/index.html#datagenerator
[4]https://www.wikidata.org/wiki/Wikidata:Main_Page
[5]https://rdf4j.org/
[6]In its Virtuoso Open-Source (VOS) edition http://vos.openlinksw.com/owiki/wiki/VOS.
[7]http://vos.openlinksw.com/owiki/wiki/VOS/VirtJenaProvider.
[8]All the parameters used to build these instances are described at http://wbsg.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/Dataset/index.html#datarules
[9]http://wbsg.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/ExploreUseCase/index.html

We created a set of queries for each template by parameterizing the template and then binding the query variables to valid values in the database. For example, to create the set of queries for the query template 1, we used query `Q3` reported in Figure 1 to obtain valid values to bind the query variables `?P`, `?PF1`, `?PF2`, and then create a set of queries produced from query template 1. This process was followed for each query template, creating eight sets with cardinality 1K.

After creating these sets of queries, we interrogated the database following a `mixed` scenario, where we issued 400 queries randomly sampled from all the templates. The queries were sampled by employing a uniform distribution to decide on the query template, and then we randomly sampled the query from the corresponding set of queries with a normal distribution. The parameters of the distribution (mean μ and standard deviation σ) were set to $\mu = |C|/2$ and $\sigma = |C|/\alpha$, where $|C|$ is the number of different queries we created for the considered class. The parameter α is used to change the standard deviation. A big α gets a small standard deviation resulting in a distribution concentrated around the mean, thus resulting in some queries repeated more frequently. Each query was repeated ten times and after excluding the minimum and the maximum, we calculated the average execution time.

We set a query timeout of five minutes for the `select` and the corresponding `construct` queries. The same value was also used as time limit for the update of the cache. We never met the timeout with the sole exception of query templates 5 and 7 for BSBM100M. From this reason, and only in that case, we ignored those query classes when sampling the queries.

Wikidata is a free and open knowledge base that acts as central storage for the structured data of its sister projects in the Wikimedia foundation [34]. It is one of the largest and most prominent collections of linked open data on the web and it is one of the most successful projects of the Wikimedia Foundation in terms of contributors [19]. Its content is available under a free license and it can be interlinked to other open data sets on the linked data web.

For the experiments, we used a 2017 version of Wikidata[10] with more than 4.7 billion triples. We employed the query logs from the University of Dresden International Center for Computer Logic. We used the Interval 2 query log[11]. From this query log we extracted the BGP$^+$ queries generated by bots (aka the "robotic" queries). We chose these queries because they can be easily converted to their `construct` version to get the provenance and present several repetitions allowing us to see the effect of CBA against the HBA lower bound baseline. For our experiments, we get the first million queries of the query log, and then we extracted the first 50K subsequent suitable BGP$^+$ queries.

Table 1 reports the average execution times obtained on BSBM using the two considered systems: HBA and CBA. As parameters we used $\alpha = 20$, $\tau = 0$ (a triple gets in the cache after it has been employed once), $l = 10$ (an epoch composed of 10 queries).

We can see that CBA speeds up the execution time for every database instance when compared to the whole database. The larger the database, the higher the improvement. In particular, we could reduce the average execution time in half for BSBM250K, or by one-third, with BSBM1M, BSBM25M, and BSBM100M. Differently, from HBA, CBA can incur a cache hit for queries never seen before. This happens when the triples in the cache overlap, even partially, with the graph

[10]In particular, the 2017-11-20 version, retrieved at the following URL: https://archive.org/details/wikibase-wikidatawiki-20171120.
[11]https://iccl.inf.tu-dresden.de/web/Wikidata_SPARQL_Logs/en

Table 1

Execution time (in ms) on different sizes of BSBM, `mixed` scenario. $\alpha = 20$. $\tau = 0$. $l = 10, \gamma = \infty$. (In the case of BSBM100M classes 5 and 7 were removed due to time constraints, the result set of class 2 is limited to 10 results since they reach cardinality of tens of thousands). Operations tagged with † are carried on offline.

	BSBM250K	**BSBM1M**	**BSBM25M**	**BSBM100M**
DB	724.41 ± 44.06	$1,531.28 \pm 130.75$	$14,394.13 \pm 1,925.79$	$23,158.99 \pm 1,607.60$
HBA	261.06 ± 45.25	445.84 ± 101.54	$4,613.31 \pm 1885.55$	$9,053.60 \pm 1,640.88$
HBA updt. time†	38.59 ± 3.86	35.0 ± 4.14	96.37 ± 8.89	931.12 ± 280.6
CBA	354.85 ± 30.53	544.85 ± 87.46	$4,567.03 \pm 1,878.72$	$7,040.75 \pm 1,333.75$
prov. time†	416.39 ± 45.47	996.36 ± 128.35	$39,632.89 \pm 5,881.16$	$24,250.71 \pm 3,425.93$
prov. time w/PC†	227.51 ± 39.83	240.15 ± 66.62	$20,517.25 \pm 4540.51$	$17,538.17 \pm 1651.31$
updt. time†	972.92 ± 131.3	865.05 ± 127.84	747.79 ± 385.67	748.43 ± 34.5
% of repetition	77.2	78.2	77.8	82.2

Table 2

Performances obtained on BSBM100M, `mixed` scenario, with different values of γ. $\alpha = 20, l = 10, \tau = 0$. The average execution on the whole database is $23,158.78 \pm 1,607.61$ ms.

BSBM100M	$\gamma = 500$	$\gamma = 1K$	$\gamma = 2K$	$\gamma = \infty$
HBA	$19,006.33 \pm 1751.32$	$15,854.61 \pm 1,711.64$	$11,017.71 \pm 1,530.49$	$7,561.2 \pm 1366.48$
CBA	$19,699.35 \pm 1715.51$	$16,650.17 \pm 1723.81$	$11,676.97 \pm 1666.4$	$7,042.82 \pm 1334.27$
hit ratio (%)	14.8	30.8	54	70.6
first time hits (%)	2.7	1.9	2.9	2.5
first time completeness (%)	42	50	51	68
completeness (%)	78	92	93	94

pattern of the new queries. In this sense, CBA can answer queries to which HBA cannot, and this brings an improvement in the average execution times that we can see for large databases where the execution time of a single query impacts more on the overall result.

We note that the time required to compute the provenance of the queries through the `construct` version and reported in the table (prov. time) may become quite big and a bottleneck for the system, even though it is carried out offline. The use of PC (see *prov.time w/PC* in the table) reduces significantly, on average, the time to get the lineage of a query processed before.

The offline time required to keep the cache and the supporting relational database updated is stable for all the sizes of the database (under 1 second). This is because the amount of data that needs to be managed (the lineage of the queries) is the same across the database sizes.

Table 2 reports the results on BSBM100M using different values for the maximum dimension of the cache ($\gamma = \{500, 1K, 2K, \infty\}$). For CBA, γ is the number of triples that are kept in the cache graph, whereas for HBA, it is the sum of the cardinalities of the result sets being cached. As it can be seen, the smaller γ, the smaller the percentage of cache hits over the set of 400 considered queries. Intuitively, there are fewer available triples and thus fewer cache hits, resulting in a worse performance with a smaller cache.

Table 2 also reports the *first time hits*, i.e., the ratio of cache hits obtained over queries seen for the first time on the total number of cache hits. The completeness of these queries cannot, in principle, be guaranteed. For this reason, we also reported the *first-time completeness*. The completeness of a query is the percentage of triples in the whole result set that are also in the cache result set; i.e., a query with 100% completeness means that the cache answers by returning

Table 3
Results on Wikidata with 5 sets of subsequent queries, composed of 10K queries each (times in ms). The reported hit and completeness ratios are computed on queries with a non-empty result set. The first-time hits and completeness ratios are computed on the same set of queries. ∅ is the percentage of queries corresponding to empty result sets. $l = 10, \tau = 0, \gamma = 1M$. Values tagged with † correspond to operations performed offline.

System	W1	W2	W3	W4	W5
DB	319.21 ± 12.73	348.49 ± 10.229	260.24 ± 9.21	91.47 ± 6.15	71.45 ± 1.5
HBA	269.23 ± 10.21	309.08 ± 8.55	235.16 ± 7.05	56.02 ± 6.18	43.02 ± 0.84
HBA updt. time†	40.17 ± 1.71	40.58 ± 0.94	44.37 ± 0.84	2332.77 ± 1593.83	48.98 ± 1.02
CBA	287.12 ± 8.4	319.96 ± 8.43	241.3 ± 7.08	106.7 ± 2.4	82.52 ± 0.42
prov. time†	133.64 ± 200.47	42.52 ± 13.12	138.83 ± 201.32	36.94 ± 4.45	133.25 ± 200.76
updt. time†	2465.14 ± 432.89	1275.06 ± 282.17	5113.34 ± 947.75	718.66 ± 168.48	643.32 ± 105.85
hit ratio (%)	48	28	22	12	45
completeness (%)	98	98	100	97	99
f.t. hits (%)	1	3.5	28	0.6	0.2
f.t. completeness (%)	49	31	100	47	55
∅ (%)	65.2	57.8	48.2	39	34

all the triples that would have been returned by evaluating the query on the whole database. Hence, the *first time completeness* is the average percentage of completeness of the result set obtained from queries that are seen for the first time; of course, for HBA, this completeness is always zero. Whereas, for CBA, some queries can be answered even though they have never been seen before; the average first-time completeness is up to 68%, showing that even though a query is seen for the first time, CBA is still returning a partially complete answer to the query if there is a hit. On the other hand, the overall completeness of the CBA answers is relatively high (92%) even with a cache cap of $2K$ triples. This shows that CBA is a viable method to return quick answers, providing a reasonable approximation of the result set. The best results for CBA are obtained with $\gamma = \infty$ reaching average completeness of 94% and the first-time completeness above 70% and an execution time smaller than that HBA achieves. Notice that $\gamma = \infty$ for BSBM100M leads to a cache with a reasonable maximum size of $3.6K$ triples (0.0036% of the DB size). Intuitively, including more data in the cache also increases the completeness in the answers to these peculiar types of queries since more triples are available to answer these queries potentially.

The dimension γ of the cache can be modified at runtime to obtain a higher number of cache hits. A bigger γ can be set to increase the cache hits, or to increase the completeness of the answers to queries seen for the first time.

Table 3 reports the results obtained on Wikidata. We used the first 50K BGP queries extracted from the considered query log, and we divided them in five "windows" (from $W1$ to $W5$) of 10K queries each. The maximum size of the CBA cache was kept to 1M triples, and that of HBA to 1M tuples. Notice that neither of the caches reached full occupation.

We notice that in windows $W1, 2, 3$, using the cache improves the average execution time and that the number of hits is between 20% and 48%. In the case of $W4$ and $W5$, the performances of CBA are lower due to a lower hit ratio (especially for $W4$) and a low average execution time (lower than 100 ms) due to the simplicity of many queries in this time window. When the queries are fast, the cache misses are particularly detrimental to the average result of CBA.

HBA is generally faster than CBA (on average) because the percentage of repeated queries is quite high, resulting in swift look-ups in the hash table. However, CBA can often answer queries never seen before, with first-time completeness ranging from 30% up to 100% for window $W3$. This result is particularly significant, showing that CBA can answer new queries with high completeness on average. Indeed, the completeness of all the queries is 97% in the worst case ($W4$) and between 98% and 100% for the other windows. This shows how CBA returns complete answers to the user in most cases.

This means that (i) the efficiency of CBA increases together with the percentage of cache hits, producing results that are close to the best-case scenario represented by HBA, and (ii) CBA can answer queries (even those never seen before) with a high level of completeness.

6. Related Work

Martin et al. [35] analyzes cached graph patterns to decide when to update the cache. Their strategy stems from the observation that only relatively small parts of a knowledge base change within a short period in typical scenarios. This approach, however, does not recognize when the same query is issued twice with minor deviations such as pattern reordering and variable renaming (*isomorphism problem*), and may store duplicate results. The proposed solution does not store the query results but a selection of previously used triples.

Yang and Wu [11] present a more sophisticated approach that caches the intermediate result of BGP SPARQL queries, i.e. Algebra Expression Trees (AETs), to speed up the subsequent operations. Our method differs since it does not cache whole results or AETs, but relevant parts of the database, thus avoiding redundancy in the data.

Lorey and Nauman [36, 37] note that machine agents typically issue queries that present similar structures and only differ in small triple pattern parts. They propose approaches to pre-fetch data by issuing never-seen-before queries. One of the limits of this greedy technique is that it cannot guarantee to find all candidate subgraph matches for a SPARQL query. Chun et al. [38] propose a proactive policy for maintaining a local cache to alleviate the expensive job of copying the data at idle time. While many works based on caching (see [35]) are content-blind, i.e. unaware of the content of cached results, Shu et al. [39] proposed instead to reuse cached queries in a content-aware fashion. The approach requires SPARQL query containment checking, which, for general SPARQL queries, is undecidable. The authors, therefore, focus on a subset of SPARQL, namely conjunctive queries with simple filter conditions (CQSF). Our method does not try to pre-fetch new data, but focuses only on data lineage and data with enough “impact”. Using provenance, we do not deal with the query containment problem [15].

Papailiou et al. [2] cache the SPARQL queries’ results in real-time by indexing the query graph patterns based on their topology. Among the limitations of this work, there is the lack of a cache replacement policy, significant storage overhead for caching the SPARQL results, lack of generalization to broader query workloads, and the complexity to put the method into use in a real-world environment.On the other hand, our technique does not require any change in the structure of an already established DBMS.

Madkour et al. [10] proposed WORQ, a workload-adaptive system that does not cache the final results, but sets of *intermediate results*, called reductions. These are then reused across multiple queries that share join patterns. Our work differentiates from WORQ because we "annotate" data with their impact rather than storing them.

The solution in this paper differs from the others since it can be seen as a layer that can be put on top of already existing systems. It is technology-independent, it relies on functionalities of the available DBMSs, without needing its own specific system.

7. Conclusions

This paper presents CBA, a cache-based architecture to reduce the answer time of SPARQL queries on RDF graphs. CBA has the advantage of being easy to implement and deploy on top of existing DBMS. It can be easily integrated with other caching and indexing systems to improve the execution time of dynamic query workloads. The triples of the RDF graph are annotated with a strike count incremented by one each time a triple appears in the lineage of a query. We calculate the impact of a triple by using a logarithmic function on the triple strike count. It guarantees a relatively fast warm-up and then a slow-growing if the triple is used over time.

CBA employs a cache to store the triples with an impact above a given threshold. The cache size is controlled by employing an LRU strategy to remove the least used triples and replace them with fresh more used ones. The overall size of the cache is a customizable parameter of the system. We tested CBA on a largely used synthetic database (BSBM) and on Wikidata by comparing CBA with a hash-based architecture, representing the best possible system employed in a controlled environment. CBA reported average answer time improvements up to one order of magnitude concerning querying the full RDF graph.

As with all cache-based solutions in the literature, let us note that CBA may return *incomplete* answers. This may happen especially with queries without the `LIMIT` clause or with big result sets. Nonetheless, CBA can be used to quickly obtain a first partial solution from a query while the system works to obtain the rest of the answer working in the background. This can be particularly useful in applications where a first answer can be useful to the users to later on refine their queries. Nevertheless, we show that CBA reaches high level of completeness in both the synthetic and the real use case, with the unique advantage of producing rather complete answers also for the queries seen for the first time.

The focus of CBA is on BGP SPARQL queries, the most common query types issued to SPARQL endpoints. In future works, we will focus on other nonmonotonic query operators, such as set difference, where the traditional definition of the lineage may not be the best suited for these applications since it would reward triples that are not useful for the cache. For this reason, we will investigate other types of provenance ranging from why-provenance to responsibility [40] and the Shapley value [41] more suited for the task.

We will also work on the management of updates on the main database. Updates may in fact make the content of the cache obsolete. In CBA, when a triple is deleted from the main database, its deletion can be seamlessly propagated to the cache. When a triple is updated or inserted, it starts with a strike count equal to 0. This value is then updated as soon as new queries are submitted. This means that the systems adapts to the update to the database, although with a cold start for new triples. Nonetheless, we employ some hash-based optimizations for fast dealing with empty queries that need to be adapted in the presence of updates.

References

[1] J. D. Fernández, J. Umbrich, A. Polleres, M. Knuth, Evaluating query and storage strategies for RDF archives, Semantic Web 10 (2019) 247–291. URL: https://doi.org/10.3233/SW-180309. doi:10.3233/SW-180309.

[2] N. Papailiou, D. Tsoumakos, P. Karras, N. Koziris, Graph-aware, workload-adaptive SPARQL query caching, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM, 2015, pp. 1777–1792. URL: https://doi.org/10.1145/2723372.2723714. doi:10.1145/2723372.2723714.

[3] M. Wylot, M. Hauswirth, P. Cudré-Mauroux, S. Sakr, RDF data storage and query processing schemes: A survey, ACM Comput. Surv. 51 (2018) 84:1–84:36. URL: https://doi.org/10.1145/3177850. doi:10.1145/3177850.

[4] W. Ali, M. Saleem, B. Yao, A. Hogan, A. N. Ngomo, Storage, indexing, query processing, and benchmarking in centralized and distributed RDF engines: A survey, CoRR abs/2009.10331 (2020). URL: https://arxiv.org/abs/2009.10331. arXiv:2009.10331.

[5] X. L. Dong, Generations of knowledge graphs: The crazy ideas and the business impact, Proc. VLDB Endow. 16 (2023) 4130–4137. URL: https://doi.org/10.14778/3611540.3611636. doi:10.14778/3611540.3611636.

[6] M. Lissandrini, D. Mottin, K. Hose, T. B. Pedersen, Knowledge graph exploration systems: are we lost?, in: Proc. of the 12th Conference on Innovative Data Systems Research, CIDR 2022, volume 22, 2022, pp. 10–13.

[7] V. Mruthyunjaya, P. Pezeshkpour, E. Hruschka, N. Bhutani, Rethinking language models as symbolic knowledge graphs, 2023. arXiv:2308.13676.

[8] A. Khan, Knowledge graphs querying 52 (2023). URL: https://doi.org/10.1145/3615952.3615956. doi:10.1145/3615952.3615956.

[9] F. Goasdoué, K. Karanasos, J. Leblay, I. Manolescu, View selection in semantic web databases, Proc. VLDB Endow. 5 (2011) 97–108. URL: http://www.vldb.org/pvldb/vol5/p097_francoisgoasdoue_vldb2012.pdf. doi:10.14778/2078324.2078326.

[10] A. Madkour, A. M. Aly, W. G. Aref, WORQ: workload-driven RDF query processing, in: The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, volume 11136 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 583–599. URL: https://doi.org/10.1007/978-3-030-00671-6_34. doi:10.1007/978-3-030-00671-6_34.

[11] M. Yang, G. Wu, Caching intermediate result of SPARQL queries, in: Proceedings of the 20th International Conference on World Wide Web, WWW 2011, ACM, 2011, pp. 159–160. URL: https://doi.org/10.1145/1963192.1963273. doi:10.1145/1963192.1963273.

[12] G. Kul, D. T. A. Luong, T. Xie, V. Chandola, O. Kennedy, S. J. Upadhyaya, Similarity metrics for SQL query clustering, IEEE Trans. Knowl. Data Eng. 30 (2018) 2408–2420. URL: https://doi.org/10.1109/TKDE.2018.2831214. doi:10.1109/TKDE.2018.2831214.

[13] R. Pichler, S. Skritek, Containment and equivalence of well-designed SPARQL, in: Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, ACM, 2014, pp. 39–50. URL: https://doi.org/10.1145/2594538.2594542. doi:10.1145/2594538.2594542.

[14] M. W. Chekol, J. Euzenat, P. Genevès, N. Layaïda, SPARQL query containment under schema, J. Data Semant. 7 (2018) 133–154. URL: https://doi.org/10.1007/s13740-018-0087-1.

doi:10.1007/s13740-018-0087-1.

[15] T. Mailis, Y. Kotidis, V. Nikolopoulos, E. Kharlamov, I. Horrocks, Y. E. Ioannidis, An efficient index for RDF query containment, in: Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, ACM, 2019, pp. 1499–1516. URL: https://doi.org/10.1145/3299869.3319864. doi:10.1145/3299869.3319864.

[16] A. Bonifati, W. Martens, T. Timm, An analytical study of large SPARQL query logs, VLDB J. 29 (2020) 655–679. URL: https://doi.org/10.1007/s00778-019-00558-9. doi:10.1007/s00778-019-00558-9.

[17] W. E. Zhang, W. Z. Sheng, Y. Qin, L. Yao, A. Shemshadi, K. L. Taylor, SECF: improving SPARQL querying performance with proactive fetching and caching, in: Proceedings of the 31st Annual ACM Symposium on Applied Computing, ACM, 2016, pp. 362–367. URL: https://doi.org/10.1145/2851613.2851846. doi:10.1145/2851613.2851846.

[18] X. Zhang, M. Wang, B. Zhao, R. Liu, J. Zhang, H. Yang, Characterizing robotic and organic query in sparql search sessions, in: Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data, Springer, 2020, pp. 270–285.

[19] S. Malyshev, M. Krötzsch, L. González, J. Gonsior, A. Bielefeldt, Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph, in: Proc. of ISWC 2018 - 17th International Semantic Web Conference, volume 11137 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 376–394. doi:10.1007/978-3-030-00668-6_23.

[20] A. Bonifati, W. Martens, T. Timm, Navigating the Maze of Wikidata Query Logs, in: Proc of the World Wide Web Conference, WWW 2019, ACM Press, 2019, pp. 127–138. doi:10.1145/3308558.3313472.

[21] J. Cheney, L. Chiticariu, W. Tan, Provenance in databases: Why, how, and where, Foundations and Trends in Databases 1 (2009) 379–474.

[22] M. Lissandrini, D. Mottin, T. Palpanas, Y. Velegrakis, Example-based search: a new frontier for exploratory search, in: Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2019, Paris, France, July 21-25, 2019, ACM, 2019, pp. 1411–1412. URL: https://doi.org/10.1145/3331184.3331387. doi:10.1145/3331184.3331387.

[23] D. Mottin, M. Lissandrini, Y. Velegrakis, T. Palpanas, New trends on exploratory methods for data analytics, Proc. VLDB Endow. 10 (2017) 1977–1980. URL: http://www.vldb.org/pvldb/vol10/p1977-mottin.pdf. doi:10.14778/3137765.3137824.

[24] J. Pérez, M. Arenas, C. Gutierrez, Semantics and Complexity of SPARQL, ACM Trans. Database Syst. 34 (2009) 1–45.

[25] M. Saleem, G. Szárnyas, F. Conrads, S. A. C. Bukhari, Q. Mehmood, A. N. Ngomo, How representative is a SPARQL benchmark? an analysis of RDF triplestore benchmarks, in: The World Wide Web Conference, WWW 2019, ACM, 2019, pp. 1623–1633. URL: https://doi.org/10.1145/3308558.3313556. doi:10.1145/3308558.3313556.

[26] L. Moreau, The foundations for provenance on the web, Found. Trends Web Sci. 2 (2010) 99–241. URL: https://doi.org/10.1561/1800000010. doi:10.1561/1800000010.

[27] Y. Ramusat, S. Maniu, P. Senellart, Semiring provenance over graph databases, in: M. Herschel (Ed.), 10th USENIX Workshop on the Theory and Practice of Provenance, TaPP, USENIX Association, 2018. URL: https://www.usenix.org/conference/tapp2018/

presentation/ramusat.

[28] G. Karvounarakis, I. Fundulaki, V. Christophides, Provenance for linked data, in: In Search of Elegance in the Theory and Practice of Computation - Essays Dedicated to Peter Buneman, volume 8000 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 366–381. URL: https://doi.org/10.1007/978-3-642-41660-6_19. doi:10.1007/978-3-642-41660-6_19.

[29] C. V. Damásio, A. Analyti, G. Antoniou, Provenance for sparql queries, in: International Semantic Web Conference, Springer, 2012, pp. 625–640.

[30] G. Flouris, I. Fundulaki, P. Pediaditis, Y. Theoharis, V. Christophides, Coloring rdf triples to capture provenance, in: International Semantic Web Conference, Springer, 2009, pp. 196–212.

[31] M. Wylot, P. Cudre-Mauroux, P. Groth, TripleProv: Efficient processing of lineage queries in a native RDF store, in: Proceedings of the 23rd international conference on World wide web, 2014, pp. 455–466.

[32] F. Geerts, U. T, G. Karvounarakis, I. Fundulaki, V. Christophides, Algebraic structures for capturing the provenance of SPARQL queries, J. ACM 63 (2016) 7:1–7:63. URL: https://doi.org/10.1145/2810037. doi:10.1145/2810037.

[33] Y. Cui, J. Widom, J. L. Wiener, Tracing the lineage of view data in a warehousing environment, ACM Trans. Database Syst. 25 (2000) 179–227.

[34] D. Vrandecic, M. Krötzsch, Wikidata: a free collaborative knowledgebase, Commun. ACM 57 (2014) 78–85. URL: https://doi.org/10.1145/2629489. doi:10.1145/2629489.

[35] M. Martin, J. Unbehauen, S. Auer, Improving the performance of semantic web applications with SPARQL query caching, in: The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, volume 6089 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 304–318. URL: https://doi.org/10.1007/978-3-642-13489-0_21. doi:10.1007/978-3-642-13489-0_21.

[36] J. Lorey, F. Naumann, Caching and prefetching strategies for SPARQL queries, in: The Semantic Web: ESWC 2013 Satellite Events, volume 7955 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 46–65. URL: https://doi.org/10.1007/978-3-642-41242-4_5. doi:10.1007/978-3-642-41242-4_5.

[37] J. Lorey, F. Naumann, Detecting SPARQL query templates for data prefetching, in: The Semantic Web: Semantics and Big Data, 10th International Conference, ESWC 2013, Montpellier, France, May 26-30, 2013. Proceedings, volume 7882 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 124–139. URL: https://doi.org/10.1007/978-3-642-38288-8_9. doi:10.1007/978-3-642-38288-8_9.

[38] S. Chun, J. Jung, K. Lee, Proactive policy for efficiently updating join views on continuous queries over data streams and linked data, IEEE Access 7 (2019) 86226–86241.

[39] Y. Shu, M. Compton, H. Müller, K. Taylor, Towards content-aware SPARQL query caching for semantic web applications, in: Web Information Systems Engineering - WISE 2013 - 14th International Conference, Nanjing, China, October 13-15, 2013, Proceedings, Part I, volume 8180 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 320–329. URL: https://doi.org/10.1007/978-3-642-41230-1_27. doi:10.1007/978-3-642-41230-1_27.

[40] A. Meliou, W. Gatterbauer, K. F. Moore, D. Suciu, The complexity of causality and responsibility for query answers and non-answers, Proc. VLDB Endow. 4 (2010) 34–45. URL:

http://www.vldb.org/pvldb/vol4/p34-meliou.pdf. doi:10.14778/1880172.1880176.

[41] D. Deutch, N. Frost, B. Kimelfeld, M. Monet, Computing the shapley value of facts in query answering, 2021. arXiv:2112.08874.