To Cache or Not To Cache: The Effects of Warming Cache in Complex SPARQL Queries

Tomas Lampo[1], María-Esther Vidal[2], Juan Danilow[2], and Edna Ruckhaus[2]

[1] University of Maryland, College Park, USA
tlampo@cs.umd.edu
[2] Universidad Simón Bolívar, Caracas, Venezuela
{mvidal,jdanilow,ruckhaus}@ldc.usb.ve

Abstract. Existing RDF engines have developed caching techniques able to store intermediate results and reuse them in further steps of the query execution process; thus, execution time is speeded up by avoiding repeated computation of the same results. Although these techniques can be beneficial for many real-world queries, the same effects may not be observed in complex queries. Particularly, queries comprised of a large number of graph patterns that require the computation of large sets of intermediate results that cannot be reused, or queries that require complex computations to produce small amounts of data, may require further re-orderings or groupings in order to make an effective usage of the cache. In this paper, we address the problem of determining a type of SPARQL queries that can benefit from caching data during query execution or warming up cache. We report on experimental results that show that complex queries can take advantage of the cache, if they are reordered and grouped according to small-sized star-shaped groups; complex queries are not only comprised of a large number of patterns, but they may also produce a large number of intermediate results. Although the results are preliminary, they clearly show that star-shaped group queries can speed up execution time by up to three orders of magnitude when they are run in warm cache, while original queries may exhibit poor performance in warm cache.

1 Introduction

SPARQL has been defined as a standard query language for RDF and several query engines have been defined to store and retrieve RDF data [3,10,11,13,14,21,28,31]. The majority of these approaches have implemented optimization techniques and efficient physical operators able to speed up execution time [3,16,21,28]. Additionally, some of these approaches have implemented structures to efficiently store and access RDF data, and have developed execution strategies able to reuse data previously stored in the cache. In this paper we focus on the study of the benefits of this last feature supported by RDF engines such as RDF-3X[22], BitMat [4], MonetDB[12], and RDFVector [19]; we study the types of queries that can benefit from caching intermediate results as well as the benefits of maintaining them in cache, or warming up cache.

The Database community has extensively studied the benefits of caching techniques to enhance the performance of queries by using recently accessed data [6,15,17,35]. On

R. Meersman, T. Dillon, and P. Herrero (Eds.): OTM 2011, Part II, LNCS 7045, pp. 716–733, 2011.

the other hand, in the context of the Semantic Web, recent publications have reported great benefits of running queries in warm or hot caches, i.e., the effects of maintaining the cache populated with valid data previously generated[4,19,22]; also, techniques to cache the most useful previous computed results have been proposed [18,33,34]. Particularly, RDF-3X and MonetDB have been reported as engines with high performance in warm cache. The behavior of RDF-3X is due to the fact that it has developed optimization and execution techniques, which in conjunction with compressed indexed structures and caching techniques, provide the basis for efficient executions of a large set of real-world SPARQL queries. In addition, these techniques benefit an efficient usage of previously loaded intermediate results in caches. However, as we will show in this paper, there is a family of queries, comprised of small-sized star-shaped groups of graph basic patterns, where further optimization and execution techniques have to be performed in order to fully exploit the RDF-3X caching features. MonetDB[35] implements a column-based storage manager and lightweight data compression techniques which provide high performance of query-intensive workloads. Additionally, MonetDB offers sophisticated cache management techniques named *in-cache processing*, to efficiently control previously cached data. Finally, MonetDB has developed a *vectorized execution* model that contrary to the traditional pipeline iterator model, is able to pass entire vectors of values through the pipeline and exploits the properties of the query engine.

Motivated by these data management features and the results reported in the literature [4,21,35], we studied the impact of the shape of SPARQL queries on the performance of RDF-3X and MonetDB, when queries are run in both cold and warm caches, i.e., when intermediate results are maintained or not in cache. We show that if these queries are rewritten as bushy plans [1] comprised of small-sized star-shaped groups, the cold cache execution time can be reduced by up to three orders of magnitude when the query is run in warm cache, while the performance of original queries may not follow the same trend. Based on these results and the fact that the RDF-3X and MonetDB are not tailored to identify or execute bushy plans comprised of small-sized star-shaped groups, we have enabled both engines to support bushy plans. First, RDF-3X engine was modified to accept plans of any shape and assign particular operators to join small-sized sub-queries in a bushy plan; we call this new version GRDF-3X. Furthermore, we implemented a translation schema to convert SPARQL bushy plans into nested SQL queries, which enforces MonetDB to execute the plan in a bushy fashion. We will describe the optimization techniques [16] that benefit the generation of bushy plans, which exploit the usage of previously cached intermediate results.

To summarize, the main contributions of this paper are the following:

- We define a family of queries that can benefit from caching intermediate results or warming up cache. These queries reduce the number of intermediate results and CPU processing, and can be rewritten as bushy plans comprised of small-sized star-shaped groups.
- We describe GRDF-3X, an extension of the RDF-3X engine able to efficiently evaluate bushy plans comprised of small-sized star-shaped groups.

[1] A bushy plan corresponds to a query plan where operands of the join operators can be intermediate results produced by other operators of the plan.

- We explain the schema translation of bushy plans into nested SQL queries that are executed against vertical partitioned tables in MonetDB.
- We provide an empirical analysis of the performance of the RDF-3X, GRDF-3X and MonetDB engines when queries are evaluated in both cold and warm caches.

This paper is comprised of five additional sections. Section 2 describes existing state-of-the-art approaches in conjunction with an analysis of the advantages and limitations of each approach. Section 3 illustrates a motivating example; section 4 presents the main features that characterize the small-sized star-shaped group queries. Section 5 presents an experimental study where we report on the performance of the small-sized star shaped group queries. Finally, we conclude in section 6 with an outlook to future work.

2 Related Work

During the last years, several RDF stores have been developed [3,10,11,13,14,21,31]. Jena [13,32] provides a programmatic environment for SPARQL; it includes the ARQ query engine and indices, which provide efficient access to large datasets. Tuple Database or TDB [14] is a persistent graph storage layer for Jena; it works with the Jena SPARQL query engine (ARQ) to support SPARQL together with a number of extensions (e.g., property functions, aggregates, arbitrary length property paths). Sesame [31] is an open source Java framework for storing and querying RDF data; it supports SPARQL and SeRQL queries. Additionally, different storage and access structures have been proposed to efficiently retrieve RDF data [7,20,29,30]. Hexastore [30] is a main memory indexing technique that exploits the role of the arguments of an RDF triple; six indices are designed so that each one can efficiently retrieve a different access pattern; a secondary-memory-based solution for Hexastore has been presented in [29]. Fletcher et al. [7] propose indexing the universe of RDF resource identifiers, regardless of the role played by the resource. Although all of the former approaches propose different strategies to speed up the execution time of RDF queries, none of them provide techniques to manage the cache or to load RDF data into resident memory which allow the observation of differences between cold and warm cache execution times.

Additionally, MacGlothlin et al. [19] propose an index-based representation for RDF documents that materializes the results for subject-subject joins, object-object joins and subject-object joins. This approach has been implemented on top of MonetDB [12] and it can exploit the Monet DB cache management system. Abadi et al. [1,2] and Sidirourgos et al. [27] propose different RDF store schemas to implement an RDF management system on top of a relational database system. They empirically show that a physical implementation of vertical partitioned RDF tables may outperform the traditional physical schema of RDF tables. In addition, any of these solutions can exploit the properties of the database manager to efficiently manage the cache.

Recently, Atre et al. [4] proposed the BitMat approach which is supported on a fully inverted index structure that implements a compressed bit-matrix structure of the RDF data. An RDF engine has been developed on top of this bit-based structure, which exploits the properties of this structure and avoids the storage of intermediate results generated during query execution. Although BitMat does not use cache techniques, it has

been shown that its performance is competitive with existing RDF engines that provide efficient cache management. Finally, RDF-3X [21] focuses on an index system, and has implemented optimization and execution techniques that support efficient and scalable execution of RDF queries. In addition, RDF-3X makes use of the Linux `mmap` system call, to load in resident memory portions of data, and thus, differences between execution time in both cold and warm caches can be observed for certain types of queries. In this paper we show how cache data management features implemented by RDF-3X and MonetDB, can be better exploited if queries are executed in a way that intermediate results are minimized.

3 Motivating Example

In this section we illustrate how the shape of a query plan can affect the performance of a SPARQL query when it is run in both cold and warm caches. SPARQL syntax resembles SQL queries where the `Where` clause is comprised of Basic Graph Patterns connected by diverse operators, e.g., join, optional, or union. Consider the RDF dataset YAGO (Yet Another Great Ontology)[2] that publishes information about people, organizations and cities around the world. Suppose a user is interested in finding groups of at most two artists who are influenced by at least one person. Figure 1 presents a SPARQL query against YAGO; the query is composed of 8 basic graph patterns connected by the join operator denoted by a "."; suppose the RDF-3X engine is used to run the query.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX yago:<http://mpii.de/yago/resource/yago>
SELECT ?A1 ?A2 WHERE
    {?A1 yago:hasFamilyName ?fn1.
    ?A1 yago:hasGivenName ?gn1 .
    ?person1 yago:influences ?A1.
    ?A2 yago:hasFamilyName ?fn2 .
    ?A2 yago:hasGivenName ?gn2 .
    ?person1 yago:influences ?A2.
    ?A1 rdf:type yago:wordnet_artist_109812338.
    ?A2 rdf:type yago:wordnet_artist_109812338.}
```

Fig. 1. SPARQL Query

Table 1 reports on the time spent by RDF-3X in a Sun Fire X4100 M2 machine with two AMD Opteron 2000 Series processors, 1MB of cache per core and 8GB RAM, running a 64-bit Linux CentOS 5.5 kernel. We can see that RDF-3X consumes 2.220 secs to evaluate the query in cold cache while the time is reduced to 0.11 secs by warming up the cache; this reduction represents 95% of the cold cache execution time, i.e., when the query is run and no data have been loaded in cache. This number is consistent with the results recently reported in the literature, where query execution time in warm cache can be reduced up to one order of magnitude with respect to cold cache execution time [4,19,22].

[2] Ontology available for download at *http://www.mpi-inf.mpg.de/yago-naga/yago/*

Table 1. Run-Time Cold and Warm Cache (secs)

Cold Cache	Warm Cache		
	Mean	Standard Deviation	Geometric Mean
2.220	0.112	0.004	0.112

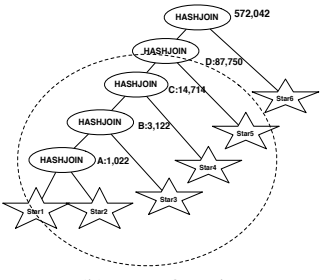
Additionally, let's consider a complex version of this query, where the user is interested in groups of at most six artists that are influenced by at least one person (Figure 2(a)). For this complex query, the behavior of the RDF-3X engine is different; the query execution time is 850.24 secs in cold cache while during warm cache, the time is reduced to 836.09 secs; thus, the savings are less than 2%. However, one can observe that almost 96% of the query execution time corresponds to optimization time in both cold and warm caches, i.e., the time to generate the plan presented in Figure 2(b).

In this plan, internal nodes correspond to the physical operator `HASH JOIN`. The leaves correspond to star-shaped groups that share exactly one variable; the `MERGE JOIN` physical operator is used to evaluate each of the stars in the query. We denote by Star_i the star-shaped group comprised of the basic graph triple patterns presented in Figure 3(a); the plan produced by RDF-3X is reported in Figure 3(b). First, the merge join is used to evaluate the join between the basic graph triple patterns; the index scan is used to access RDF-3X to recover the instantiations of the variables in the query.

Clearly, if the plan is created in way that the first pattern in the star is very selective, then the number of matched triples is reduced and the chances to locate intermediate results in cache increase. RDF-3X executes this physical plan in 48.51 secs in cold cache and 40.60 secs in warm cache; thus, if only the plan execution time is measured, then the observed savings increase to 16.30%; however, this improvement is still low. The poor performance of the RDF-3X engine in this plan can be a consequence of

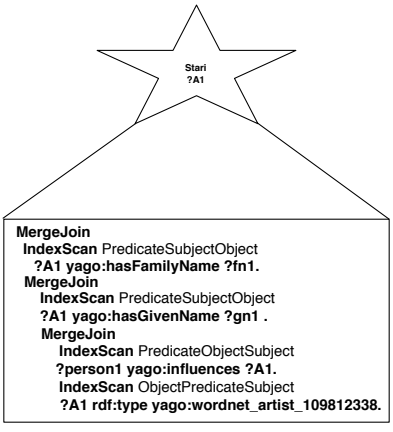
```
SELECT ?A1 ?A2 ?A3 ?A4 ?A5 ?A6 WHERE
{?A1 yago:hasFamilyName ?fn1. ?A1 yago:hasGivenName ?gn1 .
?person1 yago:influences ?A1.?A2 yago:hasFamilyName ?fn2 .
?A2 yago:hasGivenName ?gn2 . ?person1 yago:influences ?A2.
?A3 yago:hasFamilyName ?fn3. ?A3 yago:hasGivenName ?gn3 .
?person1 yago:influences ?A3.?A4 yago:hasFamilyName ?fn4 .
?A4 yago:hasGivenName ?gn4 . ?person1 yago:influences ?A4.
?A5 yago:hasFamilyName ?fn5. ?A5 yago:hasGivenName ?gn5 .
?person1 yago:influences ?A5.?A6 yago:hasFamilyName ?fn6 .
?A6 yago:hasGivenName ?gn6 . ?person1 yago:influences ?A6.
?A1 rdf:type yago:wordnet_artist_109812338.
?A2 rdf:type yago:wordnet_artist_109812338.
?A3 rdf:type yago:wordnet_artist_109812338.
?A4 rdf:type yago:wordnet_artist_109812338.
?A5 rdf:type yago:wordnet_artist_109812338.
?A6 rdf:type yago:wordnet_artist_109812338.}
```

(a) SPARQL Query



(b) RDF-3X Plan

Fig. 2. SPARQL Query and RDF-3X Plan



{?A_i yago:hasFamilyName ?fn_i.
?A_i yago:hasGivenName ?gn_i .
?person1 yago:influences ?A_i.
?A_i rdf:type yago:wordnet_artist_109812338.}

(a) Star-shaped group $Star_i$

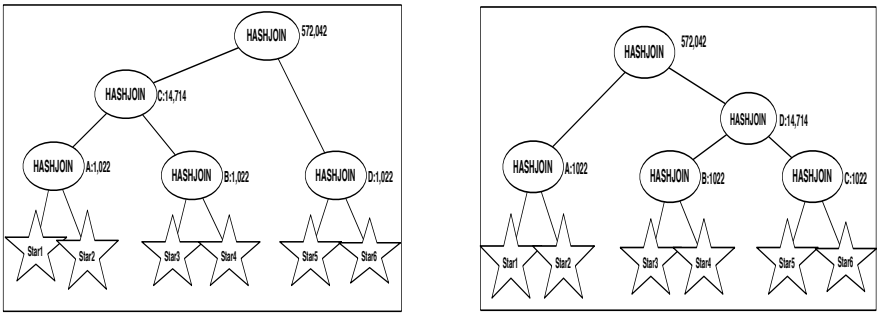
(b) RDF-3X Plan for a star-shaped group

Fig. 3. Star-Shaped Group and an RDF-3X Plan for a Star

the way this plan is evaluated. The plan is a left linear plan and generates 106,608 intermediate result triples. Particularly, the sub-plan surrounded by the circle produces 87,750 triples that may cause page faults, which degrades the performance of the RDF-3X engine.

On the other hand, if the query had been evaluated in a different fashion, the number of intermediate results could be reduced. For example, consider the plans presented in Figure 4. These plans are bushy and are comprised of star-shaped groups of small cardinality. The number of intermediate triples in any of these plans is only 17,780.

Given that any of these bushy plans maintains less triples in cache during execution time, cold cache execution time can be reduced, and the performance in warm cache is improved. The cold cache execution time for the plan in Figure 4(a) is 26.82 secs, while



(a) Bushy Plan 1

(b) Bushy Plan 2

Fig. 4. Different Bushy Plans

it consumes 14.03 secs in warm cache. Similarly, the plan in Figure 4(b) runs in 19.07 secs in cold cache and in 13.60 secs in warm cache; thus, the savings in warm cache are 47.68% and 28.68%, respectively. In this paper we illustrate the benefits of executing bushy trees in both cold and warm caches.

4 Star-Shaped Group Queries

We have developed optimization and execution techniques to support the execution of complex SPARQL queries that can be decomposed in small-sized star-shaped queries [16,28]. A star-shaped query is the join of multiple basic graph patterns that share exactly one variable or a star-shaped basic graph pattern w.r.t. ?X ($?X^* - BGP$) . This type of query is very likely to be found in real-world scenarios and can be formally defined as follows:

Definition 1 (Star-Shaped Basic Graph Pattern w.r.t. ?X, $?X^* - BGP$ **[28]).** *Each triple pattern* $\{?X\ p\ o\}$ *or* $\{s\ p\ ?X\}$ *such that* $s \neq ?X$, $p \neq ?X$ *and* $o \neq ?X$, *is a* star-shaped basic graph pattern w.r.t. ?X *denoted by* $?X^*$*-BGP. Let P and P′ be* $?X^*$*-BGPs such that,* $var(P) \cap var(P') = \{?X\}$ *then,* $P \cup P'$ *is star-shaped basic graph pattern w.r.t. ?X, i.e., an* $?X^*$*-BGP.*

Star-shaped basic graph patterns in Figure 3(a) correspond a $?A1^*$-BGP.

We have developed four different strategies (operators) that are used to retrieve and combine intermediate generated RDF triples of small-sized star shaped groups.

1. Index Nested-Loop Join (**njoin**): For each matching triple in the first pattern, we retrieve the matching triples in the second pattern, i.e., the join arguments[3] are instantiated in the second pattern through the sideways passing of variable bindings. When the data is indexed, the operator can exploit these indices with the instantiations of the second pattern to speed up the execution task. Time complexity in terms of I/O's of the **njoin** between sub-queries A and B on join variables JV, is expressed by the following formula:

$$Cost_{njoin}(A, B, JV) = Cost(A) + \#Instantiations(JV) \times Cost(B)$$

   The first term of the formula represents the cost of evaluating the outer sub-query of the join, while the second term counts the cost of executing B for each instantiation of the join variables JV. If tuples of B in the instantiations of JV that satisfy A can be maintained in cache, the number of I/O's will be reduced and in consequence, the cost of the operator will be reduced. So, the optimizer should try to select this operator, only when the number of instantiations of JV is small.
2. Group Join (**gjoin**): Given two groups, each of them is independently evaluated, and the results are combined to match the compatible mappings. The gjoin operator can take advantage of the cache, because intermediate results previously loaded in cache can be reused without the need to execute the operations required to compute them; however, the size of the results should be small to avoid page faults. Time

[3] The join arguments are the common variables in the two predicates that represent the patterns.

complexity in terms of I/O's of the **gjoin** between sub-queries A and B on join variables JV, is expressed by the following formula:

$$Cost_{gjoin}(A, B, JV) = Cost(A) + Cost(B) + 2 \times (Card(A) + Card(B))$$

The first and second terms of the formula represent the cost of evaluating the outer and inner sub-queries of the join. The third term counts the cost of storing and retrieving from disk the intermediate results produced by A and B, assuming the worst case when matches are done in hash tables previously flushed to secondary memory. In case cardinalities of A and B are minimized, results produced by A and B can be retained in cache, avoiding the cost of storing and retrieving intermediate results from disk.

3. Star-Shaped Group Evaluation (**sgroup**): the main idea is to evaluate the first pattern in the star-shaped group and identify the bindings/instantiations of the shared variable. These instantiations are used to bind the rest of the patterns in the group. This operator can be very efficient if the first pattern in the group is very selective, i.e., there are only few valid instantiations of the shared variable, and the rest of the patterns are indexed by these instantiations.
   Time complexity in terms of I/O's of the **sgroup** between basic graph triple patterns A_1, A_2, A_3, ..., A_n on shared variable JV, is expressed by the following formula:

$$Cost_{sgroup}(A_1, (A_2, \ldots, A_n), JV) = Cost(A_1) + \#Instantiations(JV) \times Cost((A_2, \ldots, A_n))$$

   The first term of the formula represents the cost of evaluating the first basic graph triple pattern; the second corresponds to the cost of executing the rest of the basic graph triple patterns with the instantiations produced by the execution of the first. The third term counts the cost of storing and retrieving from disk the intermediate results produced by executing each graph basic pattern, assuming the worst case when matches have been flushed to secondary memory. In case the number of instantiations of the shared variable JV is minimized, results produced by the execution of the rest of the basic graph patterns can be retained in cache, avoiding the cost of storing and retrieving intermediate results from disk.
4. Index Star-Shaped Group Evaluation (**isgroup**): In the case that all of the patterns in the group are indexed, the valid instantiations of each pattern can be independently retrieved and merged together to produce the output. For example, in the plan in Figure 4(a), `Star1` could be evaluated by searching the instantiations of the variable `?A1` that correspond to artists influenced by at least one person, the instantiations that have a given name, and the ones that have a given last name. These sets of instantiations are merged to produce the star-shaped group answer. This operator can benefit from running in warm cache if the number of valid instantiations to be merged, is small because computations of the join between the two basic graph patterns could be stored in cache, and reused to compute the join with the third basic graph pattern.

   Time complexity in terms of I/O's of the **isgroup** between basic graph triple patterns A_1, A_2, A_3, ..., A_n on shared variable JV, is expressed by the following formula:

$$Cost_{isgroup}(A_1, (A_2, ..., A_n), JV) = Cost(A_1) + Cost(A_2) + ... + Cost(A_n) +$$
$$2 \times (Card(A_1) + Card(A_2) + ... + Card(A_n))$$

The first n terms of the formula represent the cost of evaluating the basic graph triple patterns. The last term counts the cost of storing and retrieving from disk the intermediate results produced by evaluating the basic graph triple patterns, assuming the worst case when matches are flushed to secondary memory. In case cardinalities of the instantiations of $A_1, A_2, ..., A_n$ are minimized, intermediate results can be retained in cache, avoiding the cost of storing and retrieving intermediate results from disk.

We have implemented these four operators in our own RDF engine named OneQL [16]. Although execution time and memory usage are reduced, OneQL is implemented in Prolog and its performance cannot compete with state-of-the-art RDF engines such as RDF-3X. To fairly compare the performance of the plans comprised of these operators, we have extended the RDF-3X engine and called it GRDF-3X.

First, we modified the RDF-3X parser to consider all given plans; the original RDF-3X parser completely ignores groups and parentheses, and it flattens any input plan. Additionally, GRDF-3X exclusively assigns the RDF-3X `HASH JOIN` operator to evaluate **gjoin**, while **njoin**, **isgroup** and **sgroup** are evaluated with `MERGE JOIN`s; each basic graph pattern is evaluated by using `INDEX SCAN`. GRDF-3X reorders the patterns in a star-shaped group, but no star-groups are further identified. Based on these extensions, GRDF-3X can evaluate the plans presented in Figures 2(b) and 4, and it exploits the properties of these plans that were illustrated in Section 2. In case star-shaped groups are small-sized, the performance of RDF-3X in warm caches can be improved in several orders of magnitude.

Furthermore, we followed the *vertical partitioning approach* [1] to implement an RDF dataset as a relational database in MonetDB. For each property P, a table of two columns is defined; the first column stores the values of the subjects associated with P, while the second column stores the object values; indices are created on the two columns. A *dictionary encoding* is used to store integer keys instead of the string values of the subjects, properties, and objects. A table `Dictionary(ID,val)` maintains the encodings. SPARQL queries are translated into SQL by evaluating each triple pattern {`s P o`} in a SPARQL query as conditions `P.S` or `P.O` in the SQL query. These conditions can be placed in the `Where` clause or `Select` clause depending on the values of `s` or `o`. If `s` (res, `o`) is a constant, then the condition `P.S=id(s)` (res, `P.O=id(o)`) is added to the `Where` clause, where `id(s)` represents the encoding of `s`. If `s` (res, `o`) is a variable that also appears in another triple pattern, say, {`s P1 o'`}, and they are connected through the SPARQL operator `AND`, then the condition `P.S=P1.S` is added to the `Where` clause; similarly, if both patterns are connected by an `OPTIONAL`, a left outer join relates the conditions `P.S` and P1.S. Thus, the SPARQL operators `AND` and `OPTIONAL` are translated into a relational join and a left outer join, respectively. Finally, if both triple patterns are related through `UNION`, the condition `P.S` (res, `P.O`) is added to the `Select` clause; the SPARQL operator `UNION` is expressed as a SQL `union`. In case the SPARQL query corresponds to a bushy plan, we follow the same translation

schema for each star-shaped group of triple patterns. For a **gjoin** between sub-queries S_1 and S_2, a SQL query is created by adding to the `FROM` clause of the query, the SQL sub-queries that result from translating S_1 and S_2 with the alias s_1 and s_2; if the **gjoin** condition `JV` is on the variables $A_1, ..., A_n$, the conditions $s_1.A_1$=$s_2.A_2$ `AND ... AND` $s_1.A_n$=$s_2.A_n$ are added to the `Where` condition of the SQL query; the **njoin** is translated as the SQL join. Figure 5 illustrates the proposed translation schema.

```
SELECT ?A1 ?A2 where
    {{?A1 yago:hasFamilyName ?fn1.
    ?A1 yago:hasGivenName ?gn1 .}
    GJOIN
    {?A2 yago:hasFamilyName ?fn2 .
    ?A2 yago:hasGivenName ?gn2 .}}
```

(a) SPARQL Bushy Plan

```
SELECT s1.A1, s2.A2
FROM
      (Select HFN.S as A1
      from hasFamilyName as HFN, hasGivenName as HGN
      where HFN.S=HGN.S ) as s1,
      (Select HFN.S as A2
      from hasFamilyName as HFN, hasGivenName as HGN
      where HFN.S=HGN.S ) as s2
WHERE s1.A1=s2.A2
```

(b) SQL Query

Fig. 5. Translation Schema-SPARQL into SQL

Finally, we have developed query optimization techniques able to identify query plans comprised of small-sized star-shaped groups. These techniques have been developed in the OneQL System on top of the following two sub-components [16,23,24,25]: (a) a hybrid cost model that estimates the cardinality and execution cost of execution plans, (b) optimization strategies to identify plans comprised of small-sized star-shaped groups. The proposed optimization techniques are based on a cost model that estimates the execution time of intermediate RDF triples generated during query execution, and are able to identify execution plans of any shape. Re-orderings and groupings of the basic graph patterns are performed to identify star-shaped groups of small size. In addition, physical operators are also assigned to each join and star-shaped group. The optimizer is implemented as a Simulated Annealing randomized algorithm which performs random walks over the search space of bushy query execution plans. Random walks are performed in stages, where each stage consists of an initial *plan generation step* followed by one or more *plan transformation steps*. An equilibrium condition or a number of iterations determines the number of transformation steps. At the beginning of each stage, a query execution plan is randomly created in the plan generation step. Then, successive *plan transformations* are applied to the query execution plan in order to obtain new plans. The probability of transforming a current plan p into a new plan p' is specified by an acceptance probability function $P(p, p', T)$ that depends on a global time-varying parameter T called the *temperature*; it reflects the number of stages to be executed. The function P may be nonzero when $cost(p') > cost(p)$, meaning that the optimizer can produce a new plan even when it is worse than the current one, i.e., it has a higher cost. This feature prevents the optimizer from becoming stuck in a local minimum. Temperature T is decreased during each stage and the optimizer concludes when $T = 0$. Transformation rules applied to the plan during the random walks correspond to the SPARQL axioms of the physical operators implemented by the query engine. The axioms state properties such as: commutativity, associativity, distributivity

of gjoins over njoins, and folding and unfolding of star-shaped groups. These transformation rules are fired according to probabilities that benefit the generation of bushy plans comprised of small-sized star-shaped groups [28]. These plans usually reduce intermediate results as well as the execution time in both cold and warm caches.

5 Experimental Study

We conducted an experimental study to empirically analyze the effects of caching intermediate results during the execution of simple and complex SPARQL queries. We report on the execution time of MonetDB Apr2011 release, RDF-3X version 0.3.4 and GRDF-3X built on top of RDF-3X version 0.3.4. Particularly, we analyze the impact on the execution time performance of running bushy plans comprised of small-sized star-shaped groups in both cold and warm caches.

Benchmarking has motivated the evaluation of these query engines, and contributed to improve scalability and performance [9]. Among the most used benchmarks, we can mention: LUBM [8], the Berlin SPARQL Benchmark [5], the RDF Store Benchmarks with DBpedia[4], and the SP^{2}Bench SPARQL benchmark [26]. Similarly to existing benchmarks, we tailored a family of queries that allow us to reveal the performance of a state-of-the-art RDF engine, and we focus on illustrating the impact of the shape of query plans on the performance of the query engine in warm caches. During the definition of our benchmarks of queries, query shape, number of basic graph patterns, selectivity of the instantiations, and size of intermediate results were taken into account.

Datasets and Query Benchmark: We used the real-world ontology YAGO which is comprised of around 44,000,000 RDF triples. We developed two sets of queries[5]:

- Benchmark 1 has 9 simple queries, which are comprised of between 3 and 5 basic patterns (Figure 6(a)).
- Benchmark 2 has 9 queries, which are comprised of between 17 and 26 basic patterns (Figure 6(b)).

Additionally, we consider the LinkedCT dataset[6] which exports information of clinical trials conducted around the world; this dataset is composed of 9,809,330 RDF triples. We define a benchmark 3 comprised of 10 queries over LinkedCT; queries are composed of between 13 and 17 patterns. [7]

Evaluation Metrics: We report on runtime performance that corresponds to the *real time* produced by the *time* command of the Linux operation system. Runtime represents the elapsed time between the submission of the query and the output of the answer; optimization time just considers the time elapsed between the submission of the query and the output of the query physical plan. Experiments were run on a Sun Fire X4100 M2 machine with two AMD Opteron 2000 Series processors, 1MB of cache per core and 8GB RAM, running a 64-bit Linux CentOS 5.5 kernel. Queries in benchmark 1, 2 and 3 were run in cold cache and warm cache. To run cold cache,

[4] http://www4.wiwiss.fu-berlin.de/benchmarks-200801/

[5] *http://www.ldc.usb.ve/~mvidal/OneQL/datasets/queries/YAGO/*

[6] *http://LinkedCT.org*

[7] *http://www.ldc.usb.ve/~mvidal/OneQL/datasets/queries/LinkedCT/*

we cleared the cache before running each query by performing the command `sh -c "sync ; echo 3 > /proc/sys/vm/drop_caches"`. To run on warm cache, we executed the same query five times by dropping the cache just before running the first iteration of the query; thus, data temporally stored in cache during the execution of iteration i could be used in iteration $i + 1$. Additionally, the machine was dedicated exclusively to run these experiments.

query	#patterns	answer size
q1	4	10
q2	3	1
q3	3	4
q4	5	6
q5	3	2,356
q6	3	1,027
q7	3	5,683
q8	3	46
q9	3	1

(a) Benchmark 1

query	#patterns	answer size
q1	17	1,170
q2	21	4,264
q3	26	22,434
q4	17	238
q5	21	516
q6	26	1,348
q7	17	342
q8	21	1,220
q9	26	5,718

(b) Benchmark 2

Fig. 6. Query Benchmark Description

5.1 Performance of Star-Shaped Groups in Cold and Warm Cache

In attempting to identify the types of queries that can benefit from running in warm cache, we ran queries of Benchmark 1 in both cold and warm caches. These queries are comprised of a simple star-shaped group and were executed in RDF-3X and MonetDB. Table 2 reports on the execution time of cold cache, and the minimum value observed during the execution in warm cache; it also reports on the geometric means. We can see that both engines were able to improve performance if valid data is already loaded in cache. RDF-3X is able to improve cold cache execution times by a factor of 35 in the geometric mean when the queries are run in warm cache. MonetDB improves cold cache execution time by a factor of 31 in the geometric mean. Optimization time is negligible because the optimizer only has to reorder the patterns of the stars in each query. Additionally, the time to translate SPARQL queries into the MonetDB representation is not considered. In both cases, the majority of the execution time in warm cache was dominated by recovering the instantiations of shared variables of the star-shaped groups, that were maintained in memory during warm cache.

Then, we studied the performance of queries in Benchmark 2, which can be rewritten as bushy trees comprised of several small-sized star-shaped groups. Tables 3 and 4 report on cold cache execution times, the minimum value observed during the execution in warm cache, and the geometric means. Similarly to the previous experiment, queries were run in RDF-3X and MonetDB. Additionally, three optimized versions of the queries were run in GRDF-3X; one was created *by hand*, the other was generated by our OneQL query optimizer [16], and the last one was generated by RDF-3X. Furthermore, bushy plans that correspond to the hand-created plans were translated in SQL nested queries following the translation schema presented in the previous section.

Table 2. Benchmark 1-Run-Time RDF-3X and MonetDB in Cold and Warm Cache (secs)

Cold Caches										
	q1	q2	q3	q4	q5	q6	q7	q8	q9	Geom. Mean
RDF-3X	0.53	0.22	0.25	0.20	3.57	2.25	5.01	0.61	0.29	0.70
MonetDB	0.88	0.45	0.64	0.51	0.76	0.59	0.98	0.46	0.67	0.63
Warm Caches										
	q1	q2	q3	q4	q5	q6	q7	q8	q9	Geom. Mean
RDF-3X	0.012	0.042	0.012	0.015	0.095	0.058	0.150	0.012	0.011	0.02
MonetDB	0.01	0.01	0.02	0.02	0.04	0.08	0.05	0.01	0.01	0.02

For queries in Benchmark 2, we could observe that RDF-3X performs poorly in warm cache; cold cache times improve by nearly a factor of 1.2 in the geometric means. This behavior of the RDF-3X engine may be because the execution time of queries in Benchmark 2 is dominated by CPU-intensive processing that consumed up to 98% of the CPU time. Additionally, a large portion of the execution time was spent in query optimization and the generation of the physical plan. The reason for this is that the RDF-3X optimizer relies on a dynamic-based programming algorithm that is not able to efficiently scale up to complex queries. Finally, although plans generated by RDF-3X were comprised of small-sized star-shaped groups, they were shaped as left-linear plans, which generate a large number of intermediate results that may produce page faults.

On the other hand, we evaluated three optimized versions of the queries in Benchmark 2 in GRDF-3X: (1) optimal plans that were generated *by hand*; (2) OneQL plans that were produced by the OneQL optimizer; (3) plans generated by RDF-3X. The two first groups of optimized queries were shaped as bushy trees comprised of small-sized star-shaped groups, and ran in GRDF-3X in a bushy fashion such that the number of intermediate results was minimized. The plans generated by RDF-3X were also composed of small-sized star-shaped but combined in a left-linear tree fashion in which intermediate results were not minimal; execution times of these plans allow to illustrate RDF-3X execution time without considering optimization time. First, we could observe that the execution of the two first types of queries consumed up to 25% of the CPU time when they were run in GRDF-3X; the execution time in both cold and warm caches was reduced by up to five orders of magnitude. Also, the optimization time was negligible because GRDF-3X respected the groups in the input plan, and it only had to reorder the patterns of the stars in each query. Finally, because these two groups of plans were bushy trees comprised of small-sized star-shaped groups, the number of intermediate results was smaller; thus, intermediate results could be maintained in resident memory and used in further iterations. The GRDF-3X performance in warm cache was consistently good for hand-optimized queries; it could reduce the cold cache run time by a factor of 5 in the geometric means. For OneQL query plans, GRDF-3X reduced the cold cache run time by nearly a factor of 2.7 in the geometric means. Finally, RDF-3X generated plans exhibit a performance in warm cache that reduces execution time in cold cache by a factor of 1.88.

Furthermore, we can observe that MonetDB also performs poorly when the original query is executed in warm cache; cold cache times are improved by nearly a factor of 1.06 in the geometric means; Table 4 reports on MonetDB runtime. This observed

Table 3. Benchmark 2- RDF-3X Run-Time Cold and Warm Cache (secs)

Cold Caches										
	q1	q2	q3	q4	q5	q6	q7	q8	q9	Geom. Mean
RDF-3X	62.30	84.87	100,657.34	85.95	61.2	188,909.69	0.14	1.47	827.75	166.03
GRDF-3X (Optimal Plan)	1.60	1.80	2.34	1.22	1.38	1.36	0.99	1.05	1.75	1.45
GRDF-3X (OneQL Plan)	1.64	10.85	3.8	1.28	9.2	3.8	1.18	2.62	3,57	3.18
GRDF-3X (RDF-3X Plan)	60.92	56.16	93,010.25	60.35	59.93	183,291.76	1.34	1.7	2.64	102.68
Warm Caches										
	q1	q2	q3	q4	q5	q6	q7	q8	q9	Geom. Mean
RDF-3X	58.21	59.54	72,584.71	58.52	59.73	175,909.80	0.14	1.46	808.77	144.13
GRDF-3X (Optimal Plan)	0.34	0.26	0.93	0.14	0.31	0.17	0.12	0.31	0.69	0.29
GRDF-3X (OneQL Plan)	0.40	7.08	2.18	0.36	7.6	1.52	0.18	0.93	1.64	1.22
GRDF-3X (RDF-3X Plan)	54.42	55.42	71,231.74	58.52	50.33	140,822.38	0.25	0.29	0.64	54.34

behavior reinforces our assumption about the performance of RDF-3X in this set of queries, which are dominated by CPU-intensive processing and generate a large number of intermediate results that may produce page faults. Table 5 reports on the size in bytes of the intermediate results produced during the execution of these queries in MonetDB; these values were reported by the MonetDB tool `mclient`. The original version of q3 could not be executed in `mclient`, and the size of intermediate memory could not be computed. We can observe that optimized queries reduce the size of intermediate results by a factor of 81.24 (q3 is not considered). Additionally, we can observe that MonetDB performs very well executing the optimized queries; runtime was reduced by a factor of 1,196.76. The observed performance supports our hypothesis that optimized queries better exploit the features of both MonetDB and RDF-3X.

Finally, we conducted a similar experiment and ran queries in benchmark 3 against LinkedCT; as in previous experiments, we built an optimal bushy plan *by hand*, each optimal plan was comprised of small-sized sub-queries. In GDRF-3X sub-queries in the optimized plans were executed using the **gjoin** operator implemented in GRDF-

Table 4. Benchmark 2-MonetDB Run-Time Cold and Warm Cache (secs)

Cold Caches										
	q1	q2	q3	q4	q5	q6	q7	q8	q9	Geom. Mean
Original Query	485.30	2,993.14	3,727.33	128.57	213.03	1,751.56	576.06	3,757.61	2,622.82	1044.10
Opt Plan	0.81	0.80	1.37	0.75	1.17	0.68	0.81	0.65	1.05	0.87
Warm Caches										
	q1	q2	q3	q4	q5	q6	q7	q8	q9	Geom. Mean
Orig Plan	496.71	2,593.59	3,710.13	135.74	205.25	1,536.79	461.82	4,280.02	2,027.63	978.21
Optimal Plan	0.14	0.20	0.54	0.13	0.14	0.17	0.13	0.15	0.24	0.18

Table 5. Benchmark 2-MonetDB Size of Intermediate Results (Bytes)

	q1	q2	q3	q4	q5	q6	q7	q8	q9	Geom. Mean
Original Query	1.13E+11	2.66E+11	N/R	4.22E+10	5.57E+10	2.89E+11	7.12E+10	1.60E+11	3.24E+11	1.28E+11
Optimal Plan	1.38E+9	1.63E+9	1.99E+9	1.38E+9	1.63E+9	1.88E+9	1.37E+9	1.61E+9	1.87E+9	1.58E+9

Table 6. Benchmark 3-Execution RDF-3X Time Cold and Warm Caches (secs)

Cold Caches											
	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	Geom. Mean
RDF-3X	6.35	3.55	4.13	1,543.82	3.71	4.36	1,381.9	2.75	3.83	0.51	10.62
GRDF-3X	0.76	0.59	0.51	0.52	0.80	0.73	0.71	0.59	0.51	0.52	0.61
Warm Caches											
	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	Geom. Mean
RDF-3X	2.44	2.28	2.41	1,385.09	2.71	1.75	1,321.05	1.74	1.73	0.14	5.87
GRDF-3X	0.14	0.14	0.14	0.18	0.18	0.18	0.18	0.18	0.17	0.18	0.16

3X; SQL nested queries against vertical partitioned tables were generated to be run in MonetDB. Original queries were run in both cold and warm caches. Table 6 reports on cold cache execution times, minimum values observed during the execution in warm cache, and geometric means for RDF-3X and GRDF-3X executions; Table 7 reports on execution times for MonetDB.

We can observe that RDF-3X is able to improve cold cache execution time by a factor of 1.8 in the geometric mean when queries are run in warm cache. However, GRDF-3X performance in warm cache was consistently good for hand-optimized queries; it could reduce the cold cache run time by a factor of 3.81 in the geometric mean when the queries are run in warm cache. In addition, GRDF-3X execution times were reduced by up to four orders of magnitude in both cold and warm caches (queries $q4$ and $q7$) compared to the original query. This is because the plans were bushy trees comprised of small-sized star-shaped sub-queries, where the number of intermediate results was smaller than the original queries. Thus, intermediate results could be maintained in resident memory and used in further iterations.

Finally, we also ran these queries against MonetDB and the results are reported in Table 7; the no optimized version of q10 could not be executed because MonetDB ran out of memory. Similarly to RDF-3X, MonetDB is able to improve cold cache execution time by a factor of 1.28 in the geometric mean when the original queries are run in warm cache. However, the performance in warm cache is very good for optimized queries; the runtime is reduced by a factor of 11.65. Additionally, we can observe that optimized queries reduce runtime of original queries by a factor of 15.24.

These results provide an empirical evidence about the benefits on warm cache performance of the shape and characteristics of the plans. For simple queries, these two engines are certainly able to benefit from warming up cache; however, for queries with several star-shaped groups, the optimizers generate left-linear plans that may produce a large number of intermediate results or require CPU-intensive processing that degrades the query engine performance in both cold and warm caches. Contrary, if these queries are rewritten as bushy plans, the number of intermediate results and the CPU process-

Table 7. Benchmark 3-Execution MonetDB Time Cold and Warm Caches (secs)

Cold Caches											
	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	Geom. Mean
Original Query	3.86	4.43	4.82	13.58	13.62	6.04	12.82	13.4	13.17	N/R	8.40
Optimized Plan	2.74	2.68	2.74	11.77	11.87	4.55	11.87	11.87	11.87	N/R	6.52
Warm Caches											
	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	Geom. Mean
Original Query	0.66	0.57	0.52	0.49	0.58	0.52	0.59	0.5	0.55	0.5	0.55
Optimized Plan	0.04	0.04	0.04	0.05	0.05	0.06	0.05	0.05	0.05	0.04	0.047

ing can be reduced and the performance improves. Thus, the shape of a query plan can impact on the benefits of caching intermediate results.

6 Conclusions

We have reported experimental results suggesting that the benefits of running in warm cache depend on the shape of executed queries. For simple queries, RDF-3X and MonetDB are certainly able to benefit from warming up cache; however, for complex queries, some plans may produce a large number of intermediate results or require CPU-intensive processing that degrades the query engine performance in both cold and warm caches. We have presented a type of bushy queries comprised of small-sized star-groups that reduce the number of intermediate results and the CPU processing. In this type of queries the performance of RDF-3X and MonetDB is clearly better. These results encouraged us to extend RDF-3X with the functionality of evaluating bushy plans comprised of small-sized star-groups, and to define a translation schema to enforce MonetDB to execute queries in a bushy fashion. In the future, we plan to incorporate our optimization techniques in existing RDF engines.

References

1. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.: SW-Store: a vertically partitioned DBMS for Semantic Web data management. VLDB Journal 18(2), 385–406 (2009)
2. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 411–422 (2007)
3. AllegroGraph (2009), `http://www.franz.com/agraph/allegrograph/`
4. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In: Proceedings of the WWW, pp. 41–50 (2010)
5. Bizer, C., Schultz, A.: The berlin sparql benchmark. Int. J. Semantic Web Inf. Syst. 5(2), 1–24 (2009)

6. Bornhövd, C., Altinel, M., Mohan, C., Pirahesh, H., Reinwald, B.: Adaptive database caching with dbcache. IEEE Data Eng. Bull. 27(2), 11–18 (2004)
7. Fletcher, G., Beck, P.: Scalable Indexing of RDF Graph for Efficient Join Processing. In: CIKM (2009)
8. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. J. Web Sem. 3(2-3), 158–182 (2005)
9. Guo, Y., Qasem, A., Pan, Z., Heflin, J.: A requirements driven framework for benchmarking semantic web knowledge base systems. IEEE Trans. Knowl. Data Eng. 19(2), 297–309 (2007)
10. Harth, A., Umbrich, J., Hogan, A., Decker, S.: A Federated Repository for Querying Graph Structured Data from the Web. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 211–224. Springer, Heidelberg (2007)
11. Ianni, G., Krennwallner, T., Martello, A., Polleres, A.: A Rule System for Querying Persistent RDFS Data. In: Aroyo, L., Traverso, P., Ciravegna, F., Cimiano, P., Heath, T., Hyvönen, E., Mizoguchi, R., Oren, E., Sabou, M., Simperl, E. (eds.) ESWC 2009. LNCS, vol. 5554, pp. 857–862. Springer, Heidelberg (2009)
12. Idreos, S., Kersten, M.L., Manegold, S.: Self-organizing tuple reconstruction in column-stores. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 297–308 (2009)
13. Jena Ontology Api (2009), http://jena.sourceforge.net/ontology/index.html
14. Jena TDB (2009), http://jena.hpl.hp.com/wiki/TDB
15. Kim, S.-K., Min, S.L., Ha, R.: Efficient worst case timing analysis of data caching. In: IEEE Real Time Technology and Applications Symposium, pp. 230–240 (1996)
16. Lampo, T., Ruckhaus, E., Sierra, J., Vidal, M.-E., Martinez, A.: OneQL: An Ontology-based Architecture to Efficiently Query Resources on the Semantic Web. In: The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems at the International Semantic Web Conference, ISWC (2009)
17. Malik, T., Wang, X., Burns, R.C., Dash, D., Ailamaki, A.: Automated physical design in database caches. In: ICDE Workshops, pp. 27–34 (2008)
18. Martin, M., Unbehauen, J., Auer, S.: Improving the Performance of Semantic Web Applications with SPARQL Query Caching. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010, Part II. LNCS, vol. 6089, pp. 304–318. Springer, Heidelberg (2010)
19. McGlothlin, J.: RDFVector: An Efficient and Scalable Schema for Semantic Web Knowledge Bases. In: Proceedings of the PhD Symposium ESWC (2010)
20. McGlothlin, J., Khan, L.: RDFJoin: A Scalable of Data Model for Persistence and Efficient Querying of RDF Dataasets. In: Proceedings of the International Conference on Very Large Data Bases, VLDB (2009)
21. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. PVLDB 1(1), 647–659 (2008)
22. Neumann, T., Weikum, G.: Scalable join processing on very large rdf graphs. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 627–640 (2009)
23. Ruckhaus, E., Ruiz, E., Vidal, M.: Query Evaluation and Optimization in the Semantic Web. In: Proceedings ALPSWS 2006: 2nd International Workshop on Applications of Logic Programming to the Semantic Web and Semantic Web Services (2006)
24. Ruckhaus, E., Ruiz, E., Vidal, M.: OnEQL: An Ontology Efficient Query Language Engine for the Semantic Web. In: Proceedings ALPSWS (2007)

25. Ruckhaus, E., Ruiz, E., Vidal, M.: Query Evaluation and Optimization in the Semantic Web. In: TPLP (2008)
26. Schmidt, M., Hornung, T., Küchlin, N., Lausen, G., Pinkel, C.: An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 82–97. Springer, Heidelberg (2008)
27. Sidirourgos, L., Goncalves, R., Kersten, M.L., Nes, N., Manegold, S.: Column-store support for RDF data management: not all swans are white. PVLDB 1(2), 1553–1563 (2008)
28. Vidal, M.-E., Ruckhaus, E., Lampo, T., Martínez, A., Sierra, J., Polleres, A.: Efficiently Joining Group Patterns in SPARQL Queries. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010. LNCS, vol. 6088, pp. 228–242. Springer, Heidelberg (2010)
29. Weiss, C., Bernstein, A.: On-disk storage techniques for semantic web data are b-trees always the optimal solution? In: The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems at the International Semantic Web Conference, ISWC (2009)
30. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. PVLDB 1(1), 1008–1019 (2008)
31. Wielemaker, J.: An Optimised Semantic Web Query Language Implementation in Prolog. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 128–142. Springer, Heidelberg (2005)
32. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient RDF Storage and Retrieval in Jena2. Exploiting Hyperlinks 349, 35–43 (2003)
33. Williams, G.T., Weaver, J.: Enabling fine-grained http caching of sparql query results. Accepted ISWC (2011)
34. Yang, M., Wu, G.: Caching intermediate result of sparql queries. In: WWW (Companion Volume), pp. 159–160 (2011)
35. Zukowski, M., Boncz, P.A., Nes, N., Héman, S.: Monetdb/x100 - a dbms in the cpu cache. IEEE Data Eng. Bull. 28(2), 17–22 (2005)