Selecting Materialized Views for RDF Data

Roger Castillo and Ulf Leser

Humboldt Universtiy of Berlin
{castillo,leser}@informatik.hu-berlin.de
http://www.hu-berlin.de/

Abstract. In the design of a relational database, the administrator has to decide, given a fixed or estimated workload, which indexes should be created. This so called index selection problem is an non-trivial optimization problem in relational databases. In this paper we describe a novel approach for index selection on RDF data sets. We propose an algorithm to automatically suggest a set of indexes as materialized views based on a workload of SPARQL queries. The selected set of indexes aims to decrease the cost of the workload. We provide a cost model to estimate the potential impact of candidate indexes on query performance and an algorithm to select an optimal set of indexes. This algorithm may be integrated into an existing SPARQL query engine. We experimentally evaluate our approach on a standard query processor. We claim that our approach is the first comprehensive suggestion for the index selection problem in RDF.

Key words: SPARQL, Indexing, Index Selection, RDF, Materialized Views

1 Introduction

The index selection problem has been studied since the early 70's and its importance has been well recognized [1]. The basic principle of index selection is to find the best set of indexes for a given workload that fit into a given amount of space. Although well studied into relational databases, as far as we know, there are not approaches, which propose an automated index selection system for RDF data given a workload of SPARQL queries. An RDF data set is a collection of statements called *triples* [2], of the form *(s,p,o)* where *s* is a subject, *p* is a predicate, and *o* is an object. Each triple states the relation between subject and object. A set of triples can be represented as a directed graph where subjects and objects represent nodes and predicates represent edges connecting these nodes. The SPARQL query language is the official standard for querying RDF repositories [3]. In the relational representation, RDF triples are stored in one or more tables. A SPARQL query consists of patterns. These patterns are translated into one or more SQL queries, depending on the query and the system used. If the SPARQL query consists of more than one pattern, it is translated into an SQL query that typically contains as many joins as the query has patterns. Optimizing these joins and/or reducing their number is one of the critical issues to obtain scalable SPARQL systems.

In [4] we propose to materialize SPARQL queries and use these materialized views to speed-up queries. We target large data sets and SPARQL queries consisting of many basic graph patterns. Examples of huge data sets are, the UniProt database containing more than 600 million triples [5] or the W3C SWEO Linking Open Data Community with more than 4 billion triples [6]. In such datasets, executing a query with many graph patterns becomes a problem.

```
SELECT ?generecord ?process WHERE {
    ?pubmedrecord ?p mesh:D017966.
    ?article sc:identified_by_pmid ?pubmedrecord.
    ?generecord sc:describes_gene_or_gene_product_mentioned_by ?article.
    ?protein rdfs:subClassOf ?res.
    ?res owl:onProperty ro:has_function.
    ?res owl:someValuesFrom ?res2.
    ?res2 owl:onProperty ro:realized_as.
    ?res2 owl:someValuesFrom ?process.
    ?process <http://purl.org/obo/owl/obo#part_of> go:GO_0007166.
    ?process rdfs:subClassOf go:GO_0007166.
    ?protein rdfs:subClassOf ?parent.
    ?parent owl:equivalentClass ?res3.
    ?res3 owl:hasValue ?generecord.
}
```

Listing 1. Complex SPARQL query to retrieve all genes that are associated with both CA1 Pyramidal Neurons and the signal transduction processes [7].

Consider the query[1] in Listing 1. Executing this query on a conventional SPARQL processor results in the computation of twelve self-joins of a large triple table. However, one can safely assume properties such as, (identified_by_pmid, describes_gene_or_gene_product_mentioned_by), (onProperty, someValuesFrom), and (subClassOf, equivalentClass, hasValue) of an object are used together very often. Therefore, by materializing this information inside the system, the query could be computed with seven joins, as the materialized view would help to retrieve the information on *generecord* and *process*.

In this paper, we provide a novel approach to automate the task of selecting indexes for RDF data given a workload of SPARQL queries. We currently work with a constraint of number of indexes, but extensions to other constraint resources such as storage space is straightforward. For simplicity, in the rest of this work we refer to materialized views as indexes.

This paper is structured as follows. In Section 2 we review related work. Section 3 gives an overview of our approach and introduces algorithms to select a set of indexes. There, we also propose a cost model to evaluate candidate indexes. Section 4 provides an evaluation of our approach. Finally, we conclude in Section 5.

[1] namespaces omitted for simplicity

2 Related work

In relational database systems the problem of index selection has been continuously addressed since the early 70's [1]. In [8], Caprara et al. propose a practical solution that builds on a branch-and-bound algorithm to suggest a reduced set of candidate indexes. Chaudhuri et al. propose an index selection tool in [9]. They implement a cost-driven approach, dividing the problem into three basic stages: First, generation of a set of candidate indexes and selection of those which are more promising based on the query syntax and estimated cost. Second, optimization algorithms to evaluate sets of candidate indexes, and finally, iterative generation of multi-column indexes from the simpler "good" alternatives. An enhancement of this approach is proposed in [10]. Contrary to the previous works, this approach considers the combination of classical index structures, such as B+ trees, and materialized views to define an optimal physical design for a database system.

Similar to [10], we propose an automated selection of a set of indexes in the form of materialized views. We have in common with these previous works that we disregard the cost of updating a view, i.e., we also only consider SELECT queries in the workload. However, instead of focusing on indexing the relational representation of an RDF storage scheme, we fully exploit the RDF graph-structure for indexing. We are not indexing single attributes or triples, but fractions of queries that occur frequently in a given workload. Therefore, our approach suggests a set of native RDF/SPARQL indexes whose concepts are viable for all possible implementations of RDF stores.

In addition to the previous approaches, several efforts have been dedicated to provide data structures or algorithms to index RDF data in relational databases [11–19]. These approaches target the clever construction of indexes that are workload-unaware, i.e., they aim at achieving good scalability for any query.

In [4] we described an approach to use a predefined set of materialized SPARQL queries as indexes for RDF data. We propose the evaluation of SPARQL queries using (offline) materialized results of other queries. We refer to those materialized SPARQL queries as RDFMatView indexes. Here, we target the orthogonal problem of selecting a set of RDFMatView indexes for a workload of SPARQL queries. As far as we know, there exists no system for RDF data, which provides functionality similar to our approach. Note that in this paper we assume that the workload consists of distinct queries. An extension of the method to allow queries to appear more than once is straightforward.

3 RDFMatView Index Selection Approach

We propose the evaluation of a given workload of SPARQL queries to suggest a suitable set of RDFMatView indexes. The goal is to globally minimize the estimated response time for all queries contained in the given workload. In this section, we formally introduce all necessary concepts for this idea.

```
SELECT * WHERE {
   ?university  rdf:type ub:University;
         ub:name ?uni_name.
}
```

Listing 2. SPARQL query that returns universities and their names.

```
SELECT * WHERE {
   ?uni rdf:type ub:University;
         ub:name ?uni_name.
   ?ub_AssistantProfessor ub:worksFor ?uni;
}
```

Listing 3. SPARQL query that returns universities and their professors.

```
SELECT * WHERE {
   ?the_uni  rdf:type ub:University;
         ub:name ?uni_name.
   ?student ub:studyAt ?the_uni.
}
```

Listing 4. SPARQL query that returns universities and their students.

3.1 Workload of SPARQL Queries

Let W be a workload of n different SPARQL queries. Even when the queries in the workload are assumed to be different, there may exist some similarity between their query patterns. As example, consider a simple workload consisting of the three SPARQL queries q_1, q_2, and q_3 given in Listing 2-4, respectively. Clearly, we could materialize q_1 and reuse the result for the execution of all three queries, as q_1 is a sub-query of q_2 and q_3 (and, of course, also of q_1). On the other hand, q_1 also is very simple, and pre-computing it might not save much time for the entire workload. Suppose we were allowed to create only one index. In this case, we need to decide whether it is more advantageous to materialize q_1, gaining limited savings for all queries, or, for instance, materializing only q_3. This would only help to speed-up the query itself, but nevertheless could offer the highest total savings. As one can see, all queries are different. However, they share triple patterns. Our idea is to discover those shared triple patterns such that they can be used as indexes to improve the processing time of the workload. A candidate index set provides an initial search space for the index selection problem. Similar to index selection approaches for RDBMS (regarding a workload of SQL queries) where the attributes to index are taken from SQL queries, we extract our *index information* from the SPARQL query patterns and generate a set of indexes. Afterwards, we analyze which of them are most frequently used in the workload. Such strategy (*query containment* in the database field) requires to determine which pattern is contained in another query pattern by creating mappings between their elements, and which of them brings more savings in time for the workload. The following section describes our method in detail.

	$index_1$	$index_2$	$index_3$	**...**	$index_n$
$query_1$	x	0	0	...	0
$query_2$	0	w	0	...	t
$query_3$	y	0	r	...	0
...	...	...	...	...	...
$query_n$	z	0	0		s

Table 1. An initial set of candidate indexes. One index is generated from each query of the workload.

3.2 Candidate Indexes

In [4] we showed that it is possible to use materialized SPARQL queries to speed up the execution of other queries. We refer to a materialized SPARQL query as RDFMatView index. During query processing, the system may decide which of the existing indexes to use. First, it has to decide which indexes it may use for a given query. This task is performed by finding mappings between index and query patterns using a query containment algorithm [20] adapted for SPARQL queries. Essentially, we find all mappings between any index pattern and the query patterns by enumerating all possible cases. If a mapping exists, the index is eligible for that query. Note that, for a given index, there potentially are many different eligible mappings.

In the problem treated here, we assume a workload of SPARQL queries. From this workload, we derive a set of candidate indexes building our search space. There are different methods that may be used to provide a candidate index set. Currently, we generate one candidate index for each query, which is the query itself such that each query can be processed by using at least one index. This approach guarantees that very expensive queries are considered as potential indexes even if they are not useful for processing any other query than themselves. Therefore, the *candidate index set*, denoted as I_c, has as many indexes as the workload has distinct queries. However, for future work, we plan to consider alternative options to create candidate index sets, such as sub-queries with a minimal size constraint and query patterns with overlapping triple patterns.

An index created for a query i sometimes is also eligible for a query j. We represent this information by computing an asymmetric quadratic matrix, where the queries from the workload are the rows and the indexes are the columns. An example is shown in Table 1. A value m_{ij} in the matrix indicates which savings can be achieved by using index v_i for query q_j; if this value is 0, it means that the index is not eligible for the query. The computation of these values will be described in the following sections. The total savings of an index can be computed by summing up all values in its column.

3.3 Cost Model

To decide which indexes from the candidate index set are the most effective for the given workload, the system needs to evaluate all indexes and their influences on query processing. This decision should be made based on the expected time reduction that the usage of an index from the candidate index set brings to the execution of the queries from the workload. We refer to these savings as *gain of an index*; such gains need to be compared to the resources (such as number of indexes, storage space, or time to keep it up-to-date) allowed. Currently, our approach suggests an optimal set of indexes regarding costs and a given number of indexes (ν) since no other index information is available (indexes are generated at processing time, and therefore no offline estimations of their exact number of occurrences or processing time are possible)[2]. However, we want to emphasize that the implementation of any other resource constraint in the system is straightforward when additional index information such as number of occurrences or processing time is provided.

We estimate the gain an index offers to a query by comparing the costs it takes to execute the query with or without the index. Our cost model is purely abstract; absolute values have no meaning. We only look at the differences between different costs, which should roughly correlate with the savings in time (as shown in Section 4).

As our knowledge about the indexes is very limited, our cost model is defined on the potential number of occurrences an index could have in a given data graph. To estimate the costs, we need the size of the index pattern $|P|$, which is the number of triples in the query pattern and the size of the data graph $|G|$ (total number of triples). Assume a large triple table containing the RDF data set and a SPARQL query. Since each triple pattern from the query may potentially match all triples from the data graph, we define the cost of an query as follows.

Definition 1 (Cost of a query). *Let q be a query over a data graph G and P_q be the pattern of q. The cost $c(q)$ of q is defined as:*

$$c(q) = |G|^{|P_q|}$$

Since we build candidate indexes from a workload of SPARQL queries, we use this model to estimate costs for both queries and candidate indexes. We refer to the cost of an index as the estimated time we could save when it is precomputed; thus, high costs are better. Having a high cost means that the index pattern covers a larger number of query patterns.

Definition 2 (Saving of an index). *Let i be an index over a data graph G. The saving $s(i)$ of i is defined as:*

$$s(i) = c(i)$$

[2] An estimation of which size of ν is optimal for the workload is out of the scope of this paper

We generate a set of candidate indexes using each query pattern as an index pattern. Thus, for each query Q of the workload there exist at least one index (generated from the query pattern $P(Q)$), which can improve its processing time. However, this is the simplest case, and we look for indexes, which could be used to improve the processing time of more than one query from the workload. This particular case may imply that a query might not be completely covered by an index, i.e., there is a residual part of the query pattern, which is not covered by the index pattern. This fact requires to estimate the processing time of using the index plus evaluating the residual part of the query.

We estimate a cost for processing the residual part by looking at the number of patterns it contains and applying the cost model provided in Definition 3. Using this model to estimate costs we can now define the cost of a query when using indexes. Even though there exists a cost for retrieving the precomputed data, currently we assume real cost of an index $c(i) = 0$; if i is precomputed.

Definition 3 (Cost of a query when using an index). *Let q be a SPARQL query, i an index and let r be the residual part of q when using i, or q otherwise. The cost $c(q, i)$ of executing q using i is defined as follows:*

$$c(q,i) = \begin{cases} c(q), & \textit{if } i \textit{ not eligible for } q \\ 0, & \textit{if } |r| = 0 \\ c(r), & \textit{otherwise} \end{cases}$$

Based on the cost for executing a query with or without an index we can now define the gain an index provides when used in a query.

Definition 4 (Gain of an index). *Let q be a query and i an index. The gain of i when used in q is defined as follows:*

$$gain_q(i) = c(q) - c(q,i)$$

Note that $gain_q(i) = 0$ if i is not eligible for q.

We can now define the optimal set of indexes given a workload: It is the one set, for which the sum of the gains of the indexes in the set is the highest.

Definition 5 (Gain for a SPARQL workload). *Let W be a workload of SPARQL queries and i be an index. The gain of and index $i \in I$ in W is defined as follows:*

$$gain_W(i) = \sum_{q \in W} gain_q(i) | i \in I\}$$

Together, we have:

Definition 6 (Optimal set of indexes). *Let I be a set of candidate indexes for a workload W. Let ν be the maximum number of indexes, which can be suggested for the workload. A subset $S \subseteq I$ for speeding up W is called optimal if the following holds:*

– $\sum_{i \in S} gain_W(i)$ *is maximal and*

– $|S| \leq \nu$.

Even when our model accurately tries to capture the influence of an index in the workload, a closer look into it reveals areas of improvement. Especially, the case when two indexes are eligible for the same query, their gains are taken into account for both indexes. This behavior should be avoided since at query processing time, only one index is considered (except in the cases when indexes overlap); resulting in an overestimation of the gains. Therefore, discovering overlapping indexes is a logical next factor to consider in future research.

3.4 Enumeration of search space

Finding an optimal subset from all candidate indexes is not trivial. If there are n candidate indexes each offering a specific gain, choosing the optimal set under a space constraint is NP-Complete, as shown in [1] by reduction on the Knapsack problem. This also holds for a uniform space model as in our case. However, there are fast approximation algorithms that have provable quality. Specifically, we propose the use of a greedy heuristic [21], which sorts the items (indexes) in decreasing order of estimated value. We then select indexes in decreasing order until the ν parameter is reached. This algorithm guarantees that our solution is bounded with a value of at least $gain_W(J) \leq \frac{gain_W(I)}{2}$, where $J \subseteq I$ and $gain_W(I)$ is the maximal gain that can be achieved from the candidate index set using ν indexes.

4 Evaluation

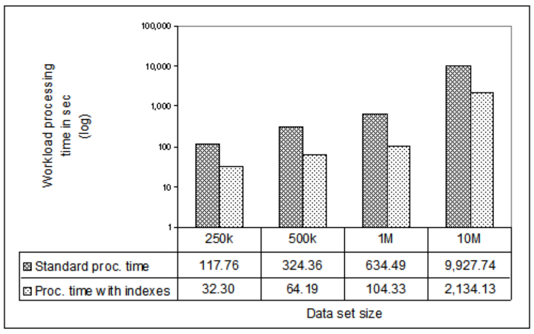
This section describes the evaluation of our approach using the Berlin SPARQL Benchmark [22]. We use the ARQ/Jena RDF Storage System (version 2.5.7) [23] and the RDFMatView approach [4] on Postgres 8.2 as framework.

4.1 Dataset and queries

To evaluate the performance of our approach, we generate a set of nine indexes and compare the workload processing time (using indexes) against standard query processing (without indexes) over four datasets with 250k, 500k, 1M, and 10M triples respectively. Additionally, we evaluate our approach by creating three sets of indexes suggested by our approach containing 3, 4 and 5 indexes respectively. As in the previous test, these sets are used to process the entire workload and their processing time is compared to the workload processing time when using five randomly generated index configurations (each configuration consists of 3 sets containing 3, 4 and 5 indexes). We evaluate these six configurations over an RDF dataset containing 500K triples. Our datasets are based on e-commerce use case information and were created by using the data generator provided in [22]. Based on the set of queries provided in [22], we derive 18 queries as workload. We transformed the query patterns into simple graph patterns (the only form of patterns the current RDFMatView approach implementation can cope with) and removed bindings to variables. Bound variables incur extremely high selectivity resulting in the retrieval of only a handful of triples. Such queries are well supported by existing index structures and do not require the type of join-optimization that is achieved with the RDFMatView approach; therefore, performance gains would be only marginal.

4.2 Results

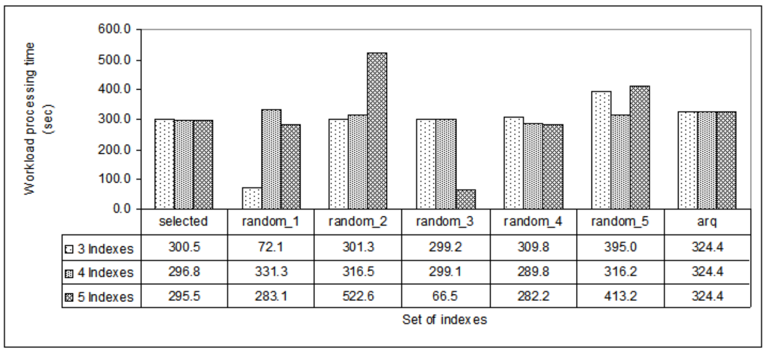
Using our approach, we generate a set of indexes for each dataset and evaluate the workload using the RDFMatView approach and plain ARQ (without indexes). To compute the workload processing time, all queries were executed 5 times and average execution times are reported. Figure 1 illustrates the processing time for each workload over the four datasets. For each dataset, the set of suggested indexes contains 9 indexes. These indexes could be used to process 10 queries from the workload respectively. In all scenarios, the workload processing time dramatically improves in comparison to standard query processing.



	250k	500k	1M	10M
▩ Standard proc. time	117.76	324.36	634.49	9,927.74
▨ Proc. time with indexes	32.30	64.19	104.33	2,134.13

Fig. 1. Workload processing time without using indexes compared to workload processing time using nine suggested indexes. The graphic illustrates that for each dataset, there is a large gain of processing time for the given workload when using the set of indexes suggested by our approach.

To verify the goodness of our approach, we generate three optimal sets of indexes (based on our cost model) consisting of 3, 4, and 5 indexes. We also create five random index configurations with the same number of indexes and evaluate the workload using the RDFMatView system and plain ARQ (without indexes). With this configuration we want to stress that our suggested sets of indexes are a suitable indexing solution for the given workload. Figure 2 shows that all sets suggested by our system improve the processing time of standard query processing (without indexes). In fact, the processing time resulting from using our selected indexes is better than the 66% of the processing time when using random index sets and ARQ. However, it also shows that some random index configurations are quite better than our index selection (see random_1 and random_3 with 3 and 5 indexes respectively). A closer look into the workload reveals that some queries generate large numbers of matches when they are queried against the dataset independent from their number of triple patterns, incurring in costly processing time. Clearly, our cost model does not consider these cases since indexes are evaluated regarding their number of triple patterns and no assumption about real number of occurrences is foreseen. In [4] has been proved



	selected	random_1	random_2	random_3	random_4	random_5	arq
▢ 3 Indexes	300.5	72.1	301.3	299.2	309.8	395.0	324.4
▦ 4 Indexes	296.8	331.3	316.5	299.1	289.8	316.2	324.4
▩ 5 Indexes	295.5	283.1	522.6	66.5	282.2	413.2	324.4

Fig. 2. Workload processing time using six different index configurations over a 500K triples dataset. Each configuration contains three sets with 3,4 and 5 indexes respectively. The workload was processed using indexes selected by our approach, randomly selected indexes and without indexes (standard ARQ).

that covering a large number of patterns using as few indexes as possible significantly improves the processing time. These savings in time are due to the number of joins required to answer the SPARQL query, which are drastically reduced using this approach. To perform this process, RDFMatView requires overlapping indexes[3], which allow an optimal covering of the query patterns. Since our index selection approach suggests indexes regarding the complete query pattern for each query, it may occur that those indexes do not overlap at all. In queries where the number of covered patterns is smaller than the number of residual patterns, the final processing time may increase depending on the number of partial results resulting from the covered and uncovered patterns using RDFMatView query processing. Thus, the selection of potential overlapping indexes which could be used to process queries from the workload and a more accurate estimation of the number of query occurrences are natural next factors to consider in future work.

[3] indexes with triple patterns in common

5 Conclusions and future work

We introduced and developed a system to analyze SPARQL queries, which returns a set of RDFMatView indexes such that their application in the query processing improves the entire workload processing time. Candidate indexes are evaluated by a cost model regarding their potential number of occurrences in a given dataset. Results show that indexes selected using our approach can dramatically decrease the workload processing time. Furthermore, we analyzed and compared the processing time of the workload using six different index configurations. Each configuration consists of 3 index sets with 3, 4 and 5 indexes respectively. One of these configurations is suggested by our system and five are randomly generated. Upon analysis, indexes selected by our approach outperform randomly selected indexes in most cases by assuring a suitable indexing solution for the given workload. Up to now, our approach is restricted to queries containing only a basic graph pattern and without filters, modifiers and blank nodes. We currently work with a constraint on the number of indexes, but extensions to other constraints like storage space are straightforward. Currently, we are working on novel ideas for optimization of the index selection process analyzing not only the given queries but also generating potential indexes based on sub patterns of the query patterns (assuming a minimum pattern size). Another interesting improvement is to generate statistics using fixed predicates, for instance $count(?x, p, ?y)$, which can be used as a fine-grained cost model for each triple pattern. We believe that such an approach should improve the quality of the suggested indexes.

References

1. Comer, D.: The difficulty of optimum index selection. ACM Trans. Database Syst. **3**(4) (1978) 440–445
2. Manola, F., Miller, E.: RDF Primer (February 2004) W3C Recommendation.
3. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF (April 2008) W3C Recommendation.
4. Castillo, R., Leser, U., Rothe, C.: RDFMatView: Indexing RDF Data for SPARQL Queries. Technical report, Humboldt University of Berlin (2010)
5. Dataset, U.R.: http://dev.isb-sib.ch/projects/uniprot-rdf/
6. Project, W.S.C.: Linking open data on the semantic web. http://esw.w3.org/topic/sweoig/taskforces/communityprojects/linkingopendata/
7. Stenzhorn, H., Srinivas, K., Samwald, M., Ruttenberg, A.: Simplifying access to large-scale health care and life sciences datasets. In: ESWC. (2008) 864–868
8. Caprara, A., Fischetti, M., Maio, D.: Exact and approximate algorithms for the index selection problem in physical database design. IEEE Transactions on Knowledge and Data Engineering **7**(6) (1995) 955–967
9. Chaudhuri, S., Narasayya, V.R.: An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In: VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1997) 146–155

10. Agrawal, S., Chaudhuri, S., Narasayya, V.R.: Automated Selection of Materialized Views and Indexes in SQL Databases. In: VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (2000) 496–505
11. Groppe, J., Groppe, S., Ebers, S., Linnemann, V.: Efficient Processing of SPARQL Joins in Memory by Dynamically Restricting Triple Patterns. In: Proceedings of the 24th ACM Symposium on Applied Computing (ACM SAC 2009), Honolulu, Hawaii, USA, ACM (March 8 - 12 2009) 1231–1238
12. Groppe, S., Groppe, J., Linnemann, V.: Using an Index of Precomputed Joins in order to speed up SPARQL Processing. In Cardoso, J., Cordeiro, J., Filipe, J., eds.: Proceedings 9th International Conference on Enterprise Information Systems (ICEIS 2007 (1), Volume DISI), Funchal, Madeira, Portugal, INSTICC (June 12 - 16 2007) 13–20
13. Jinghua Groppe, S.G., Kolbaum, J.: Optimization of SPARQL by Using coreSPARQL. In Cordeiro, J., Filipe, J., eds.: Proceedings of the 11th International Conference on Enterprise Information Systems, Volume DISI,(ICEIS 2009), Milano, Italien, INSTICC (Mai 6 - 10 2009) 107–112
14. Olaf Hartig, R.H.: The SPARQL Query Graph Model for Query Optimization. In: ESWC2007. (2007)
15. Matono, A., Amagasa, T., Yoshikawa, M., Uemura, S.: An Indexing Scheme for RDF and RDF Schema based on Suffix Arrays. In: Proceedings of SWDB 2003. (2003) 151–168
16. Udrea, O., Pugliese, A., Subrahmanian, V.S.: GRIN: A Graph Based RDF Index. In: AAAI. (2007) 1465–1470
17. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proc. VLDB Endow. **1**(1) (2008) 1008–1019
18. Yan, X., Yu, P.S., Han, J.: Graph indexing based on discriminative frequent structure analysis. ACM Trans. Database Syst. **30**(4) (2005) 960–993
19. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proc. VLDB Endow. **1**(1) (2008) 647–659
20. Halevy, A.Y.: Answering queries using views: A survey. The VLDB Journal **10**(4) (2001) 270–294
21. Dantzig, G.: Linear Programming and Extensions. Princeton University Press (August 1998)
22. Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. International Journal On Semantic Web and Information Systems - Special Issue on Scalability and Performance of Semantic Web Systems, 2009 (2009)
23. ARQJena: ARQ - A SPARQL Processor for Jena. http://jena.sourceforge.net/ARQ/ (2010)