SpiderStore: Exploiting Main Memory for Efficient RDF Graph Representation and Fast Querying

Robert Binna, Wolfgang Gassler, Eva Zangerle, Dominic Pacher, Günther Specht
Databases and Information Systems, Institute of Computer Science
University of Innsbruck, Austria
{firstname.lastname}@uibk.ac.at

ABSTRACT

The constant growth of available RDF data requires fast and efficient querying facilities of graph data. So far, such data sets have been stored by using mapping techniques from graph structures to relational models, secondary memory structures or even complex main memory based models. We present the main memory database *SpiderStore* which is capable of efficiently managing large RDF data sets and providing powerful and fast SPARQL processing facilities. The *SpiderStore* storage concept aims at storing the graph structure in main memory without performing any complex mappings. Therefore it exploits the natural web-structure of RDF by using fast and random access to main memory. The abandonment of additional mappings or meta-information therefore leads to a significant performance gain compared to other common RDF stores.

Categories and Subject Descriptors

H.3.2 [**Information Storage and Retrieval**]: Information Storage; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; H.2 [**Database Management**]: Query Processing

General Terms

Performance, Algorithms, Design, Experimentation

Keywords

RDF, Main Memory, Database, RDF Store, Triple Store, SPARQL, Query Processing

1. INTRODUCTION

Due to the ever growing, vast amounts of RDF data, the storage of this data is mostly realized on persistent media – either in a native store or within a relational database system. Storing huge amounts of RDF data in relational databases has been facilitated by most of the popular RDF stores and Semantic Web frameworks as very big ontologies did not fit into main memory so far.

However, the relational database model has not been intended for the storage of large graph structures. Current solutions for the storage of graphs in relational tables feature a mapping between the graph structure and the relational tables of the database system [2, 13, 20]. This mapping significantly slows down both the storing and the querying process, which causes a significant loss of performance. Another important factor is that the sequential access characteristics of persistent memory requires multiple indices for fast data access, which basically requires a duplication of the stored data. Storing all information on persistent media also features the disadvantage of very costly I/O operations. This is especially limiting when performing join operations which feature potentially large intermediate results which have to be joined. Depending on the type of mapping and the queries executed, these operations over a huge number of triples on secondary media can be an expensive task in terms of execution time or memory consumption.

Recent developments in the area of high-performance main memory OLTP and OLAP processing databases have minimized the I/O bottleneck by moving the whole database system into main memory which has become appropiate in terms of space capacity even for large datasets. Systems facilitating such an architecture – like VoltDB [1] or HyPer [12] – allow to process several 100,000 transactions per second while providing full ACID properties. Though they have moved to main memory, these approaches are still based on the relational paradigm and therefore suffer from an impedance mismatch when it comes to mapping graph based data into a tabular layout. Hence these approaches have to deal with costly join operations and potentially large intermediate results when processing graph structures.

We present *SpiderStore*, a main memory based RDF database which overcomes these limitations and exploits fast random read and write operations of main memory and provides time and space efficient SPARQL query processing facilities.

The remainder of this paper is structured as follows. Section 2 outlines the memory layout of *SpiderStore*. Subsequently, Section 3 is concerned with the fast and efficient processing of SPARQL queries on the proposed main memory database. Section 4 features the evaluation of the *SpiderStore* prototype and Section 5 discusses related work. The paper is concluded by a summary and the description of future work in Section 6.

Workshop on Semantic Data Management (SemData@VLDB) 2010, September 17, 2010, Singapore.

2. MEMORY LAYOUT

In the following section we sketch the lightweight storage layout of *SpiderStore*, which is optimized for fast and efficient SPARQL query processing on RDF graphs. In order to utilize the nature of main memory architecture, the *SpiderStore* approach stores a graph natively as a set of nodes and pointers (edges). Due to the fact that main memory is more expensive than disk based memory and therefore limited, a very lightweight layout – without any complex mappings, index structures or additional meta-information – is required. These conditions are satisfied by the following memory layout which is based on two basic building blocks:

Nodes within an RDF graph are either subjects, objects or predicates, where each subject is connected to an object by an appropriate edge, such that the predicate annotating the edge again is a node. Furthermore, the identifier of each node is stored within the node itself.

Edges connect two nodes, one serving as subject and one as object. Edges are implicitly realized by pointers from the subject node to the object node.

To be able to browse through the graph structure in order to answer queries efficiently, the memory layout is optimized for graph traversal operations. This is implemented by storing all edges belonging to a certain node in a dense memory block. This implies that scanning all edges of a certain node only requires a linear scan of a continuous memory block. Within this block, all edges of the same predicate are clustered and linked to the predicate node itself. Furthermore, all predicates are stored as nodes as well. The RDF graph data model consists of a directed graph where all edges point from the subject to the object. However, our approach features bidirectional edges. This is due to the fact that the query processing performance can be increased significantly by the possibility of browsing through the graph in any direction. The benefit of such a structure – where subjects, objects and predicates are nodes connected by a pointer structure – is that the graph can be traversed efficiently in any direction from any starting point within the graph. An additional index which consists of all predicates and lists of its sources (subjects) even facilitates to start the traversal at a predicate. The access to nodes via their identifier is guaranteed within a time complexity of $\mathcal{O}(log(n))$ by using an additional index. This index is used for the fast and efficient conversion between strings (e.g. URIs) and main memory nodes.
Figure 1 sketches the memory model for the storage of RDF data. The illustrated example features the following nodes: `Node 1` (subject) is connected by `Node 2` (predicate) to the object `Node 3`. `Node 3` itself is also a predicate of the connection between `Node 4` and `Node 1` where `Node 4` is the subject and `Node 1` is the object. The following listing shows the example data from Figure 1 represented in triple notation:

```
...
<Node 1> <Node 2> <Node 3>
<Node 4> <Node 3> <Node 1>
...
```

The estimated memory consumption (number of bytes m) for a given RDF data set stored in the described memory

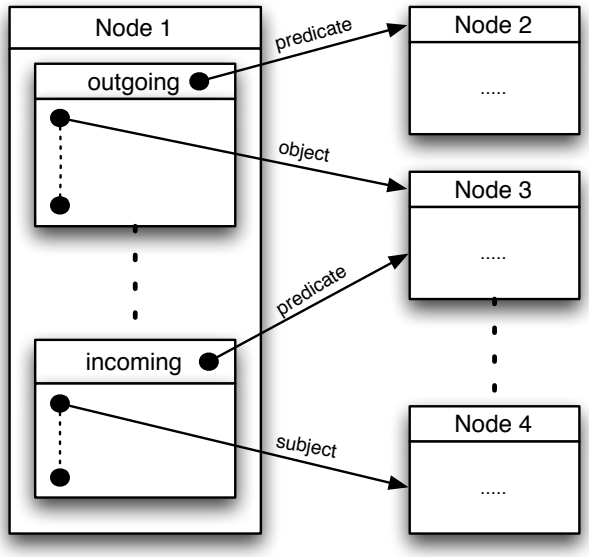

Figure 1: RDF Store Main Memory Layout

layout can be calculated by applying this formula:

$$m = (\#nodes * 5 + \#edges * 3) * sizeof(pointer)$$

where $\#nodes$ is the overall number of nodes and $\#edges$ is the number of edges (triples) within the RDF data set. The subformula $\#edges * 3$ contains the space estimate for the incoming, the outgoing edge (bidirectional) and the link between the edge and the corresponding entry node. For each *node*, space for a pointer to all its incoming and a pointer to all its outgoing edges, together with their degree and a pointer from the string-index (dictionary) has to be allocated. The default pointer size on a 64bit architecture is 8 byte. The formula does not consider the strings (identifiers) itself which are stored in a dictionary-like string index. Strings (URIs, literals) are stored only once which results in an additional memory consumption of $\sum_{i=0}^{n} length\_in\_byte(unique\_string_i)$, where n denotes the number of unique strings within the data set.

3. QUERY ENGINE

In the *SpiderStore* system, the benefits of storing all graph data within the main memory are exploited for the query process within the stored data.
Secondary memory-based RDF stores prefer breadth-first search because the huge amount of random memory accesses would be disadvantageous. This approach naturally leads to intermediate results. Due to the typically high volume of RDF data sets, intermediate results tend to be very large [3], which is a limiting factor for the performance of the search algorithm as writing and reading of intermediate results is very expensive in terms of time and space capacities. In the context of a main memory store, the limited amount of memory available is a crucial factor when handling intermediate results.
A memory efficient solution is accomplished by applying depth-first search instead of breadth-first search. As argued by Gardarin *et al.* [9], depth-first search is an efficient operation for traversing paths as long as all data is kept in

memory, which is true for *SpiderStore* by design. Furthermore, they point out that by using depth-first search no intermediate results are required to traverse existing paths. Therefore, the complexity of a query is only restricted by the amount of main memory used to store the overall result. For queries which do not facilitate any postprocessing operations like sorting or grouping, *SpiderStore* provides the results in a stream-based way and therefore does not require any additional memory.

The *SpiderStore* query engine splits SPARQL queries into so-called "restriction triples", which are similar to Basic Graph Patterns [16]. The result set of a query is defined by all variable assignments (paths) satisfying a given set of restrictions. Therefore, these restriction triples can be used to traverse the subgraph that contains the result data in depth-first order. Figure 2 illustrates an exemplary query which selects all scientists who were born in a Swiss city and whose doctoral advisor was born in a German city. Nodes which are marked by a "?" represent variables and are bound by the query engine during the graph walk. The solid lines in the Figure represent the structure of the subgraph. The dot dashed lines represent the order in which the triple patterns are applied to the stored RDF graph by the query engine.

To determine the execution order of triples, selectivity heuristics of basic graph patterns as described in [16] are exploited. *SpiderStore* determines the execution order by implicitly storing basic statistical data. This data contains information about the number of property instances and the number of incoming and outgoing edges grouped by the property of each node. Hence this statistical data does not need to be precomputed and can be used to determine the execution order by ordering the restriction triples based on their selectivity. Therefore, the restriction triples containing the most selective nodes, edge or combination of these, are applied first in order to keep the amount of nodes which have to be processed as small as possible.

The processing order of the restrictions is crucial as the restrictions may share variables, which have to be unified and therefore have to be processed in the correct order. The fact that *SpiderStore* connects all nodes bidirectionally is very beneficial for the computation as restriction triples can be processed in left-to-right or right-to-left order. After this execution order is defined, a set of seed nodes is determined, which marks the starting point(s) of the search process. Depending on the selectivity, these seeds can either be a restricted subject or object node (e.g. "scientist" in the example query in Figure 2) or nodes which are connected to a very selective predicate. During the next steps, the restrictions are applied in a depth-first order. For each matching restriction, the next restriction in order is pushed onto an execution stack. A result is found if all restrictions available are pushed onto this stack and are therefore satisfied by the current path. Subsequently, the last restriction is popped from the stack and the next edge or node at the current position is processed. The iteration continues until the stack is empty and no more nodes are available for processing. The variables defined within a query are tracked by using a shared variable assignment which is used during the graph traversal to allow the unification of variables. Every time a result is detected, the current state of these variable assignments is returned as a result. The result is streamed to the client or saved for further postprocessing (e.g. filter conditions and grouping).

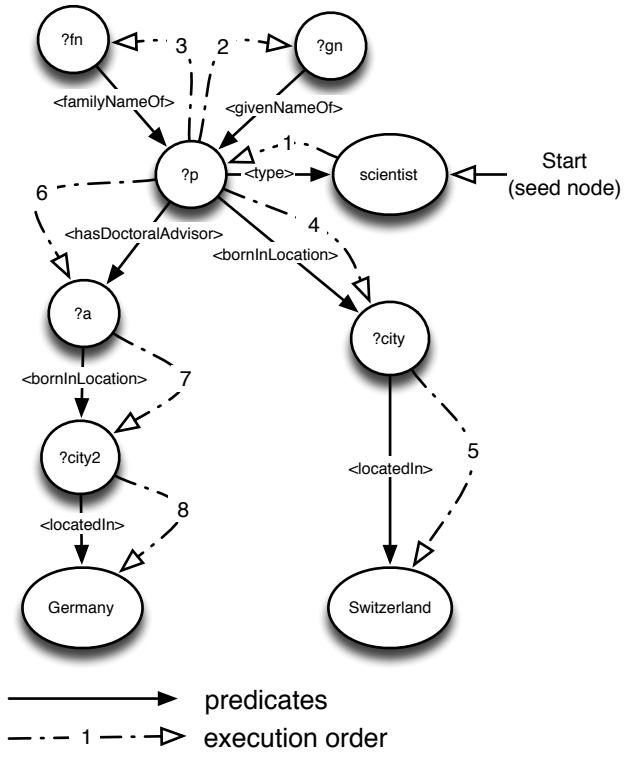

Figure 2: Example Query

4. EVALUATION

All experiments were conducted on a server equipped with two Intel Xeon L5520 Quad Core CPUs, 2.27 GHz, Linux kernel 2.6.18, CentOS, 64-bit architecture and 48 GB main memory.

4.1 Data Sets

For the evaluation we used two data sets: YAGO [17] and DBpedia [4]. The YAGO ontology is a huge knowledge repository based on Wikipedia and WordNet data. YAGO consists of 39,193,669 triples (93 predicates, 33,951,502 unique subjects and objects). Based on the this data set, we executed the SPARQL queries, which already served as a benchmark for the RDF-3X approach [13] on the data sets. The DBpedia project [4] is concerned with the extraction of triple information contained within Wikipedia pages and infoboxes. The data set contains a total of 94,839,012 triples (39664 predicates, 23,090,848 unique subjects and objects). For the DBpedia data set, we used the project's SPARQL example queries, extended them and added further complex queries in order to cover common query types (similar to the YAGO query set). An exact formulation of the queries used for the evaluation can be found in the appendix.

4.2 Evaluated Systems

For the evaluation of *SpiderStore*, we compared it against the main memory RDF storage Sesame version 2.3.1 [7] without an inferencer enabled. Additionally, we compared it to the native secondary memory stores Jena TDB version 0.8.6 [20] and a standard installation of RDF-3X version 0.3.4 [13]. All systems were granted a maximum of 16 GB main memory for their computations. However, we were

Query	A1	A2	A3	B1	B2	B3	C1	C2	geom. mean
SpiderStore	**0.0012**	0.0765	**0.0126**	**0.0518**	**0.0055**	0.9984	0.3558	**0.0196**	**0.0352**
Sesame	0.0849	0.0387	0.1072	0.2252	0.1591	0.3765	dnf	0.0386	0.3200
RDF-3X	0.0184	**0.0166**	0.0360	0.0787	0.0381	**0.0929**	**0.3377**	0.0534	0.0522
Jena TDB	0.3090	0.9000	125.375	2.5100	2.9310	4.8552	dnf	8.9253	7.1286
#results	1	7	147	56	3,714	74	1,842	1	

Table 1: Query Runtimes on the YAGO data set (in seconds).

only able to set up Sesame for our tests by granting it 30 GB of main memory. All systems feature single threaded query engines and only use one out of 16 available cores when executing queries. For the experiments of secondary storage systems, the warm cache measurements were conducted by executing the queries five times without dropping the caches and taking the best result for the evaluation.

4.3 Evaluation Results

The query execution times for the YAGO data set can be seen in Table 1, where "dnf" marks a query which was aborted after a run time of 10 minutes without having obtained a result. As for the YAGO data set, *SpiderStore* is able to compute the query results significantly faster than the other systems in 5 out of 8 queries. *SpiderStore* performs better than all other systems with regards to the geometric mean of all query execution times. The secondary memory system RDF-3X performs significantly better than the main memory system Sesame. However, the lightweight storage structure of *SpiderStore* performs better than RDF-3X on warmed caches. Queries consisting of triple patterns which contain many variables can result in a big amount of paths which have to be validated. The order in which the restriction triple of such a query are processed is crucial. This fact is beneficial for RDF-3X as it features a very mature query optimization engine, which is heavily based on statistics, which are precomputed during the import process. *SpiderStore* currently contains a very naive optimizer and is therefore not able to exploit such facts, which results in slower query execution compared to RDF-3X on query A2, B3 and C1.
The query execution times for the experiments on the DBpedia data set are listed in Table 3. The experiments on the DBpedia data set showed that *SpiderStore* clearly surpasses all other systems in terms of execution time, even though the query engine optimizer is very limited and not very mature. For example projections on certain variables are not taken into account during the processing of queries. Therefore, the system tries to satisfy all variables occurring within the query, even if they are eventually not even asked for in the query, which again leads to an computational overhead during the execution of queries. Furthermore, queries featuring "union" or "optional" statements are currently split up into two queries, executed sequentially and the results are joined in the end.
In addition to the query execution time measurements, the import times for the compared systems was measured. We define the import time as the time required for the systems to load all data and build all the required indices. Table 2 lists the import times for the compared systems. *SpiderStore* outperforms the other systems as due to the implicit statistic information no additional statistics have to be computed. However, we have to note that in terms of restart time disk based systems, like RDF-3X and Jena TDB, naturally outperform main memory systems as all data structures are kept on secondary memory and therefore do not have to be built on every startup of the system, whereas for main memory systems, the import has to be performed on every startup.

System	YAGO	DBpedia
SpiderStore	7:50	29:41
Sesame	25:47	53:51
RDF-3X	16:15	48:21
Jena TDB	38:52	53:49

Table 2: Import Times (in minutes)

5. RELATED WORK

Basically there are two approaches for storing RDF data in triple form: (i) storing triples in a traditional relational database or (ii) using a native triple store. The first, very popular approach maps all RDF triples onto tables of relational databases. This can either be realized by using a potentially very big triple table or by splitting the triples according to their predicates and storing all triples featuring the same predicate in a separate table (property tables). The worst case scenario for property tables is obtaining one table per predicate. However, the clustered property table approach tries to solve this problem by clustering predicates into tables based on their co-occurrence within the data set. These different approaches have been evaluated and benchmarked in [18].
There are several approaches for main memory RDF stores: Brahms [11] is a main memory RDF store, which aims at high-performance association discovery based on variable-length queries on the RDF graph. GRIN [19] is concerned with the creation of an RDF index optimized for long path queries. In contrast, *SpiderStore* aims at efficiently answering all kinds of SPARQL queries and is not specialized on any particular query type.
As for the popular Semantic Web Frameworks, Sesame [7] provides relational storage, in-memory storage and a native storage engine. The Sesame in-memory storage engine uses a list of quadruples called statements where each statement consists of a subject-, predicate-, object- and a context

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	geom. mean
SpiderStore	**0.0378**	**0.00005**	**0.0002**	**0.0287**	**0.1127**	**0.0018**	**0.0005**	**0.00272**
Sesame	80.9006	0.0010	0.0009	0.2184	0.2341	0.0541	0.0099	0.0572
RDF-3X	0.1197	0.0338	0.0273	0.0288	0.6778	0.0351	0.0058	0.0460
Jena TDB	dnf	dnf	94.246	1.128	1.623	6.644	0.200	13.5192
#results	2	4	21	1,560	1,172	1	2	

Table 3: Query Runtimes on the DBpedia data set (in seconds).

node. Each node maintains statement lists of the node's occurrences as subject, predicate, object and context. Jena [20] also provides both a relational store (SDB) and a native triple store (TDB). Virtuoso [8] can be facilitated on top of a native triple store (Virtuoso Triple Store) or on top of a relational database system (Virtuoso RDF Views). Abadi *et al.* [2] and Sidirourgos *et al.* [15] exploited column-wise storage of RDF data. YARS2 [10] facilitates native storage and makes use of six (relational) indices for subject, predicate, object and context. The RDF-3X system [13] is also based on extensive indices which are heavily compressed. RDF-3X features indices for all possible orderings and subsets of subject, object and predicate. It also provides extensive heuristics about the stored data, which are further exploited for the processing of queries. Furthermore, Neumann and Weikum proposed join order optimizations based on sideways information passing and selectivity heuristics [14]. However, such extensive index structure are neither feasible nor required when dealing with main memory. The BitMat project [3] provides a lightweight RDF index for main-memory RDF stores which is based on a BitMat, a bitcube index responsible for a compressed storage and querying of triples. All of these systems have already been compared extensively in [6, 13]. As for the optimization of RDF storage and querying, Stocker *et al.* [16, 5] focus on the optimization of Basic Graph Patterns based on selectivity heuristics.

6. CONCLUSION AND FUTURE WORK

In this paper we presented a lightweight in-memory storage layout for RDF graphs which was implemented in a first prototype called *SpiderStore*. This layout provides fast and efficient in-memory RDF querying and storage without having to perform any complex mapping. Our experiments showed that a simplistic storage layout, which is specifically designed for graph data is able to outperform common stores based on the relational model. The performance gain is based on (1) the lightweight storage model which represents edges as memory pointers and therefore allows direct node jumps in the graph, (2) implicit statistics about the selectivity which can be used to optimize the processing order and (3) depth-first query processing which renders unnecessary intermediate results and expensive join chains. Future work on *SpiderStore* will include the optimization of the query execution order based on implicit statistical information which currently relies on a naive algorithm. Furthermore we will improve the memory management and layout regarding writing facilities.
newline

7. REFERENCES

[1] Voltdb. Technical report, March 2010. `http://www.voltdb.com/_pdf/VoltDBOverview.pdf`.

[2] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[3] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 41–50, New York, NY, USA, 2010. ACM.

[4] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. *The Semantic Web*, pages 722–735, 2007.

[5] A. Bernstein, C. Kiefer, and M. Stocker. OptARQ: A SPARQL optimization approach based on triple pattern selectivity estimation. *Rapport technique, Department of Informatics, University of Zurich*, 2007.

[6] C. Bizer and A. Schultz. The berlin SPARQL benchmark. *International Journal On Semantic Web and Information Systems*, 5(1), 2009.

[7] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. *The Semantic WebISWC 2002*, pages 54–68, 2002.

[8] O. Erling and I. Mikhailov. RDF Support in the Virtuoso DBMS. *Networked Knowledge-Networked Media*, pages 7–24.

[9] G. Gardarin, J. Gruser, and Z. Tang. Cost-based selection of path expression processing algorithms in object-oriented databases. In *Proceedings of the international conference on very large data bases*, pages 390–401, 1996.

[10] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A federated repository for querying graph structured data from the web. *The Semantic Web*, pages 211–224.

[11] M. Janik and K. Kochut. Brahms: A workbench RDF store and high performance memory system for semantic association discovery. *The Semantic Web–ISWC 2005*, pages 431–445.

[12] A. Kemper and T. Neumann. Hyper: Hybrid OLTP & OLAP high performance database system. Technical Report TU-I1010, TU Munich, Institute of Computer Science, Germany, May 2010.

[13] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.

[14] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 627–640, New York, NY, USA, 2009. ACM.

[15] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *Proceedings of the VLDB Endowment*, 1(2):1553–1563, 2008.

[16] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 595–604, New York, NY, USA, 2008. ACM.

[17] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A Core of Semantic Knowledge. In *16th international World Wide Web conference (WWW 2007)*, New York, NY, USA, 2007. ACM Press.

[18] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking database representations of RDF/S stores. *The Semantic Web–ISWC 2005*, pages 685–701, 2005.

[19] O. Udrea, A. Pugliese, and V. Subrahmanian. GRIN: A graph based RDF index. In *Proceedings of the National Conference on Articial Intelligence*, volume 22, page 1465. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[20] K. Wilkinson, C. Sayers, H. Kuno, D. Reynolds, et al. Efficient RDF storage and retrieval in Jena2. In *Proceedings of SWDB*, volume 3, pages 7–8. Citeseer, 2003.

APPENDIX

A. QUERIES

A.1 YAGO data set

A1: select ?gn ?fn where { ?gn ⟨givenNameOf⟩ ?p. ?fn ⟨familyNameOf⟩ ?p. ?p ⟨type⟩ scientist. ?p ⟨bornInLocation⟩ ?city. ?p ⟨hasDoctoralAdvisor⟩ ?a. ?a ⟨bornInLocation⟩ ?city2. ?city ⟨locatedIn⟩ Switzerland. ?city2 ⟨locatedIn⟩ Germany. }
A2: select ?n where { ?a ⟨isCalled⟩ ?n. ?a ⟨type⟩ actor". ?a ⟨livesIn⟩ ?city. ?a ⟨actedIn⟩ ?m1. ?a ⟨directed⟩ ?m2. ?city ⟨locatedIn⟩ ?s. ?s ⟨locatedIn⟩ United States. ?m1 ⟨type⟩ movie. ?m1 ⟨producedInCountry⟩ Germany. ?m2 ⟨type⟩ movie. ?m2 ⟨producedInCountry⟩ Canada. }
A3: select distinct ?n ?co where { ?p ⟨isCalled⟩ ?n. {?p ⟨type⟩ actor } union { ?p ⟨type⟩ athlete }. ?p ⟨bornInLocation⟩ ?c. ?c ⟨locatedIn⟩ ?s. ?s ⟨locatedIn⟩ ?co. ?p ⟨type⟩ ?t. filter(?t reaches politician via ⟨sub-ClassOf⟩) }
B1: select distinct ?n1 ?n2 where { ?a1 ⟨isCalled⟩ ?n1. ?a1 ⟨livesIn⟩ ?c1. ?a1 ⟨actedIn⟩ ?movie. ?a2 ⟨isCalled⟩ ?n2. ?a2 ⟨livesIn⟩ ?c2. ?a2 ⟨actedIn⟩ ?movie. ?c1 ⟨locatedIn⟩ England. ?c2 ⟨locatedIn⟩ England. filter (?a1 != ?a2) }
B2: select ?n1 ?n2 where { ?p1 ⟨isCalled⟩ ?n1. ?p1 ⟨bornInLocation⟩ ?city. ?p1 ⟨isMarriedTo⟩ ?p2. ?p2 ⟨isCalled⟩ ?n2. ?p2 ⟨bornInLocation⟩ ?city. }
B3: select distinct ?n1 ?n2 where { ?n1 ⟨familyNameOf⟩ ?p1. ?n2 ⟨familyNameOf⟩ ?p2. ?p1 ⟨type⟩ scientist. ?p1 ⟨hasWonPrize⟩ ?award. ?p1 ⟨bornInLocation⟩ ?city. ?p2 ⟨type⟩ scientist. ?p2 ⟨hasWonPrize⟩ ?award. ?p2 ⟨bornInLocation⟩ ?city. filter (?p1 != ?p2) }
C1: select distinct ?n1 ?n2 where {?n1 ⟨familyNameOf⟩ ?p1. ?n2 ⟨familyNameOf⟩ ?p2. ?p1 ⟨type⟩ scientist. ?p1 ?i1 ?city. ?p2 ⟨type⟩ scientist. ?p2 ?i2 ?city. ?city ⟨type⟩ ⟨site⟩. filter (?p1 != ?p2) }
C2: select distinct ?n where { ?p ⟨isCalled⟩ ?n. ?p ?i1 ?c1. ?p ?i2 ?c2. ?c1 ⟨type⟩ ⟨village⟩. ?c1 ⟨isCalled⟩ London. ?c2 ⟨type⟩ ⟨site⟩. ?c2 ⟨isCalled⟩ Paris. }

A.2 DBpedia Data Set

prefix rdf: ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#⟩
prefix foaf: ⟨http://xmlns.com/foaf/0.1/⟩
prefix dbpedia2: ⟨http://dbpedia.org/property/⟩
prefix skos: ⟨http://www.w3.org/2004/02/skos/core#⟩
prefix dbo: ⟨http://dbpedia.org/ontology/⟩

Q1: select ?name ?name2 ?place where { ?person dbo:birthPlace ?place. ?person foaf:name ?name . ?place skos:subject ⟨ http://dbpedia.org/resource/Category:European_Capitals_of_Culture⟩ . ?person2 dbo:birthPlace ?place . ?person2 foaf:name ?name2 . ?person skos:subject ?type1 . ?person2 skos:subject ?type2 . ?type1 skos:broader ⟨http://dbpedia.org/resource/Category:Musicians⟩ . ?type2 skos:broader ⟨http://dbpedia.org/resource/Category:Musicians⟩ . filter(?name != ?name2) }
Q2: select ?name1 ?name2 where {?person foaf:name ?name1 . ?person2 foaf:name ?name2 . ?person dbpedia2p:influences ?person2 . ?person2 dbpedia2:awards ⟨http://dbpedia.org/resource/Time_100:_The_Most_Important_People_of_the_Century⟩ .}
Q3: select ?name where { ?vehicle skos:subject ?cat . ?vehicle dbpedia2:transmission "6-speed manual" . ?vehicle dbpedia2:name ?name . ?vehicle dbo:manufacturer ?man . ?man dbo:location ?location . ?location dbo:leaderName ⟨http://dbpedia.org/resource/Alfred_Lehmann⟩ . ?cat skos:broader ⟨http://dbpedia.org/resource/Category:Luxury_vehicles⟩.}
Q4: select distinct ?person ?person2 where {?person ?z ⟨http://dbpedia.org/resource/Category:IBM_Fellows⟩. ?person ?prop ?value. ?person2 ?prop2 ?value2. ?person2 ?prop3 ⟨http://dbpedia.org/resource/Category:IBM_Fellows⟩. filter (?person != ?person2)}
Q5: select distinct ?scientist ?doctor where { ?scientist rdf:type ⟨http://dbpedia.org/ontology/Scientist⟩. ?scientist ⟨http://dbpedia.org/ontology/doctoralStudent⟩ ?doctor. ?doctor ?property ?value. ?scientist ?prop2 ?value. }
Q6: select distinct ?player where {?s dbpedia2:name ?player. ?s rdf:type ⟨http://dbpedia.org/ontology/SoccerPlayer⟩. ?s dbpedia2:position "Goalkeeper". ?s ⟨http://dbpedia.org/property/clubs⟩ ?club. ?club ⟨http://dbpedia.org/ontology/capacity⟩ "32609" . ?s ⟨http://dbpedia.org/ontology/birthPlace⟩ ?place. ?place dbpedia2:populationCensus "49138831".}
Q7: select ?name ?birth ?description ?person ?x where {?person dbo:birthPlace ⟨http://dbpedia.org/resource/Berlin⟩. ?person skos:subject ⟨http://dbpedia.org/resource/Category:German_musicians⟩. ?person dbo:birthDate ?birth. ?person foaf:name ?name. }