Using Graph Summarization for Join-Ahead Pruning in a Distributed RDF Engine

Sairam Gurajada[†] Stephan Seufert[†] Iris Miliaraki[†] Martin Theobald[‡]

[†]{gurajada, sseufert, miliaraki}@mpi-inf.mpg.de
Max-Planck Institute for Informatics
Saarbrücken, Germany

[‡]martin.theobald@ua.ac.be
University of Antwerp
Antwerp, Belgium

ABSTRACT

The need for scalable and efficient RDF stores has seen a high demand recently. Many efficient systems, both centralized and distributed, have been proposed. Since a row-oriented output is required by SPARQL, most of the current systems rely on relational joins. One of the problems with relational joins, though, is a performance bottleneck imposed by the generation of large intermediate relations which could be avoided by using more accurate data and pruning statistics. To address this problem, recently several systems have been proposed that employ bisimulation-based graph summaries – adopted from XML indexing – over large RDF graphs in order to facilitate join-ahead pruning. In this paper, we discuss a different, locality-based, graph summarization approach for RDF data and highlight its utilization for join-ahead pruning in a distributed SPARQL engine. Based on our recently developed TriAD engine, we present a detailed comparison of processing techniques for these graph summaries over the synthetic LUBM benchmark.

1. INTRODUCTION

The Resource Description Framework (RDF) and the SPARQL query language[1] are two recent standards recommended by the W3C for representing and querying linked data on the Web. RDF has become the main standard for semantic data and meanwhile found a wide adoption in the Database as well as the Semantic Web communities. With the increasing number of both commercial and non-commercial organizations, which actively publish RDF data, the amount and diversity of openly available RDF repositories is growing at an unprecedented pace. DBpedia[2], for example, which serves as the main hub for the Linked Open Data[3] (LOD) initiative, currently consists of more than 1 billion RDF triples. As of 2011, the entire LOD cloud already consisted of more than 31 billion RDF triples which are distributed across more than 300 LOD sources.

Consequently, and in response to this explosion of RDF data that is available on both the surface and the deep Web, much research effort has been invested recently in the development of scalable, both centralized and distributed, techniques for indexing RDF data and for processing SPARQL queries. Among the centralized approaches, native RDF stores like Jena, Sesame, HexaStore [12], SW-Store [1], MonetDB-RDF [11], RDF-3X [7, 8], BitMat [2] and TripleBit [13] have been carefully designed to keep pace with the increasing scale at which RDF collections are available. Several distributed architectures [10, 15, 4, 14] have been proposed in the recent past for the scalable management of large RDF collections.

Due to the row-oriented output required by the SPARQL 1.0 and 1.1 standards, most of these systems directly follow a relational approach for joining triples. One of the main challenges that persist with relational (i.e., join-based) approaches is that – even for selective queries – often large intermediate relations are generated. This is mainly due to sub-optimal join orders determined by the plan generator which may rely on insufficient or inaccurate statistics. Several methods have been proposed to overcome this challenge. One approach, used for example in RDF-3X, is Sideways Information Passing (SIP). SIP shares variable bindings across different join operators in a query plan, thus aiming to prune unnecessary intermediate (i.e., "dangling") triples as early as possible. On the other hand, systems like [2] (centralized) and [14] (distributed) follow a two-staged approach to reduce the number of dangling triples. The first stage, called pruning stage, is performed over the RDF data graph by using a light-weight graph exploration technique that finds the possible bindings for the query variables. These bindings are then joined using a relational approach, which results in the final in row-oriented output. A drawback of the former runtime SIP approach is the particular way in which bindings are communicated among join operators. This limits its benefit to only certain types of queries (e.g., "star" queries), but remains less effective for path-like or mixed star/path queries which are very common in SPARQL.

Recently, a few approaches [6, 9, 16] have been proposed to address the limitations of two-staged RDF systems. These approaches use bisimulation-based data summaries in the first stage in order to prune dangling triples. Although the generated summaries are relatively small compared to the actual RDF graph, they aim to retain the principal structural characteristics of the original RDF data graph. Thus, these approaches have been shown to perform better than the two-staged approaches for a wide class of queries. One of the main problems with bisimulation-based summaries is that with an increasing depth (to increase pruning), the number of possible graph synopses grows exponentially, and thus the summary information may quickly grow as large the original data. Thus, these approaches limit bisimulation to a few levels, which in turn limits the overall pruning effectiveness. To overcome these challenges, our recently proposed TriAD [3] engine employs a locality-

[1] http://www.w3.org/TR/rdf-sparql-query/
[2] http://dbpedia.org
[3] http://linkeddata.org

SWIM'14, June 27, 2014 - Snowbird, Utah, USA.

http://dx.doi.org/10.1145/2630602.2630610 .

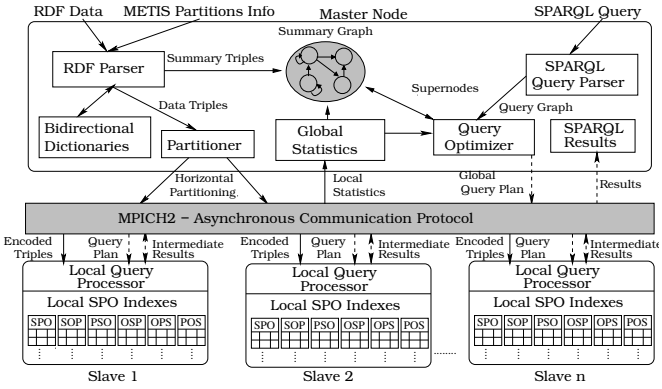

Figure 1: TriAD system architecture

based approach towards summarizing large RDF data graphs. In this paper, we thus provide a detailed discussion of the summarization framework employed TriAD, and we provide a detailed experimental comparison of three processing techniques for the summary graph, which are based on *1-hop* and *full graph exploration*, and on *relational joins*, respectively.

2. TRIAD ARCHITECTURE

In this section, we provide an overview the TriAD architecture, which encompasses graph summarization to facilitate join-ahead pruning in a distributed setting. TriAD is a distributed shared-nothing RDF engine which in principle follows a classic master-slave architecture. All inter-node communication is based on an asynchronous Message Passing protocol that is coupled with a parallel query processing framework. The details of the architecture are shown in Figure 1. The master node in TriAD performs the task of RDF data and query optimization, and it performs a first-stage processing of queries over the summary graph. The slaves perform the second-stage processing of queries over the RDF data graph to obtain detailed pruning information from the summary graph in the first stage. Details about index construction, query optimization and processing in TriAD can be found in [3]. In the following, we focus on the graph summarization and its integration within the TriAD query processing workflow.

3. GRAPH SUMMARIZATION

We argue that join-ahead pruning is one of the main factors that influence the performance of a relational system, which holds especially also for an RDF engine. Here, we discuss a join-ahead pruning technique via graph summarization. A summary graph for a given RDF data graph retains the principal characteristics of the data graph in more concise form. By summarizing the data, large portions of irrelevant data items can be pruned by first querying the summary graph and then processing the query over the remaining, pruned data graph. Before discussing the different approaches of summarizing RDF graphs, we first formally define the RDF summary as follows.

Definitions

DEFINITION 1. *An **RDF data graph** $G_D(V_D, E_D, L, \phi_D)$ is a directed, labeled multi-graph where V_D is the set of data nodes, E_D is the set of directed edges connecting the nodes in V_D, L is the set of edge and node labels, and ϕ_D is a labeling function with $\phi_D : V_D \cup E_D \rightarrow L$ s.t. $\forall v_i, v_j \in V_D$, $v_i \neq v_j$, it holds that $\phi_D(v_i) \neq \phi_D(v_j)$.*

DEFINITION 2. *An **RDF summary graph** $G_S(V_S, E_S, L, \phi_S)$ for a given RDF data graph $G_D(V_D, E_D, L, \phi_D)$ again is a labeled, directed multi-graph where each node $v \in V_S$, with $v \subseteq V_D$, called **supernode**, is a subset of nodes in V_D, and each edge $e \in E_S$, called **superedge**, connects two supernodes in V_S, and $\phi_S : E_S \rightarrow L$ maps each superedge $e \in E_S$ to a label in L.*

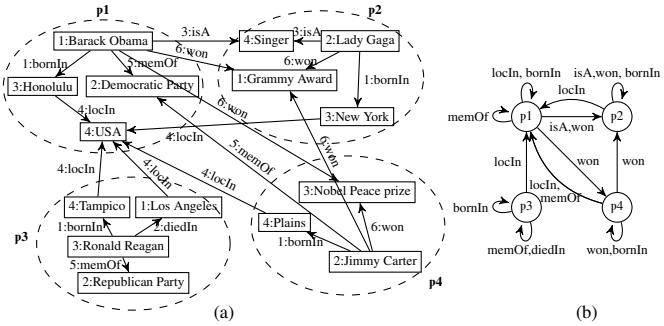

Figure 2: (a) RDF data graph G_D with locality-based partitioning; and (b) summary graph G_S for G_D

Bisimulation vs. Locality-based Summary

There are many ways in which a summary graph for an RDF graph can be generated. Recently proposed centralized systems such as [6, 9, 16] extend the idea of bisimulation-based summaries which have been used for XML data to RDF graph summarization. Bisimulation-based summaries are generated by grouping structurally similar nodes of an RDF graph into a single supernode. The structural similarity of two nodes is recursively defined as a function of the similarity of their neighborhood. Although bisimulation captures the structural information of an RDF graph, the size of the generated summary is highly dependent on the extent of the neighborhood (or recursion) considered. Thus, in the worst case, a full bisimulation may result in summaries that are as large as the original RDF graph. In practice, the summary is generated by limiting the depth to a few hops. This way, one can restrict the summary to a smaller size, however at the cost of pruning effectiveness, thus making this kind of summaries effective only for queries where predicates are constants and all the subjects/objects are variables. In real world applications, queries with constant subjects/objects are commonly seen and to effectively handle them, a locality-based summarization is more helpful than one obtained via bisimulation.

For the RDF graph shown in Figure 2 (a), its locality-based summary is shown in (b), where the nodes that are in the neighborhood are grouped together into a supernode. Thus with the constant subject/object k in the query, only the supernodes that in the neighbourhood of the supernode that contain k are considered, which helps in pruning irrelevant portions of the data graph.

Generating & Indexing Summaries

In TriAD, we use a non-overlapping graph partitioning algorithm (like METIS [5]) to generate a locality-based summary. Although METIS is one of the most scalable graph partitioning tools available, the sheer size of RDF graphs prevents us to use METIS directly on the RDF data graph. Instead, to reduce the size of the RDF data, we first remove non-influential edges in the RDF graph, i.e., edges that connect string literals. This way, the size of the graph is reduced considerably without affecting the min-cut optimality used in METIS. The partition information ($\langle node, part\_id \rangle$) obtained from METIS is then used to generate the summary triples. If $\langle s, p, o \rangle$ is a data triple and p_1, p_2 are the partition ids of s, o, then $\langle p_1, p, p_2 \rangle$ forms the summary triple for $\langle s, p, o \rangle$. The generated summary triples are then indexed at the master node. To support an

efficient exploratory search over this summary index, we index the summary triples in adjacency-list form (i.e., using two PSO, POS permutations). These indexes are then lexicographically sorted to facilitate efficient lookups via binary search. In addition to these indexes, we also compute various statistics over the summary graph, which are used for determining the best graph exploration order during query processing.

Choosing the Number of Partitions

A difficult challenge in generating the locality-based summary lies in determining the optimal number of partitions that minimizes the overall query cost of our two-staged processing approach. In TriAD, we formulate this problem as a convex optimization problem with the goal of identifying the optimal number of partitions, thus reducing the overall query cost also in a distributed setting. We arrived at a simple equation for determining the number of partitions $|V_S|$, which is expressed in terms of the number of triples $|E_D|$, the average degree d of each node in G_D, and the number of physical compute nodes n.

$$V_S := \sqrt{\frac{\lambda |E_D|}{d \times n}} \tag{1}$$

Here, λ is a tuning parameter that is obtained from an empirical analysis of the hardware setup and expected query workload. More details about the convex formulation and the tuning parameter λ can be found in [3].

4. JOIN-AHEAD PRUNING

As described earlier, query processing in TriAD is performed in two-stages. The first stage – the pruning stage – is performed over the summary graph. The goal of this stage is to prune dangling triples by identifying the bindings for the join variables in the query, and to later on use these bindings to generate results via the relational joins. Note that, in TriAD, we use a summary graph and the bindings that we obtain are supernode bindings which contain a contiguous block of triples in our actual SPO indexes. For finding the bindings, we perform a graph exploration over the summary graph. The reason behind choosing graph exploration over a conventional join-based approach is that, in this first stage, we aim to just detect the possible variable bindings. Further, by using an exploration-based approach, we can avoid generating large intermediate relations which would not easily be possible with relational joins.

Graph Exploration

Unlike in Trinity.RDF [14], the graph exploration algorithm we use performs a full exploration rather than a simpler 1-hop graph exploration. Under a full graph exploration, we add a supernode binding to a join variable only if there exists at least one binding for all the other variables in the query, such that all join conditions are satisfied. This way, many false-positive bindings can be pruned in comparison to a 1-hop exploration. The disadvantage of a full graph exploration is that it is more expensive compared to a 1-hop exploration but, since the summary graph is relatively small compared to the data graph, the disadvantages are compensated with the overall savings in query processing times.

Example. Consider the RDF data shown in Figure 2 together with the following SPARQL query:

R_1 : `?person <bornIn> ?city.`
R_2 : `?city <locatedIn> USA.`
R_3 : `?person <won> ?prize.`
R_4 : `?prize <hasName> ?name.`

For a fixed exploration order R_1, R_2, R_3, R_4, the full graph exploration works as follows. We start with relation R_1 and find the first bindings for join variables `?person` and `?city` . Then the binding for `?city` is checked with the condition that it is located in `USA` in relation R_2. Further, we check in R_3 whether `?person` binding has won any `?prize`, and the `?prize` binding has at least one name `?name` binding in R_4. If, at any stage, any such binding does not satisfy the query constraints, we propagate this information backwards to prune also the other variables' bindings . This back-propagation is not performed in the 1-hop exploration used in [14]. For the example graph shown in Figure 2(a), the obtained bindings for the 1-hop and full graph exploration are as follows.

1-Hop Exploration

`?person`: *Barack Obama, Jimmy Carter, Lady Gaga*
`?city`: *Honolulu,* ***Tampico****, Plains, New York*
`?prize`: *Nobel Peace Prize, Grammy Award*
`?name`: *"Nobel Peace Prize", "Grammy Award"*

Full Exploration

`?person`: *Barack Obama, Jimmy Carter, Lady Gaga*
`?city`: *Honolulu, Plains, New York*
`?prize`: *Nobel Peace Prize, Grammy Award*
`?name`: *"Nobel Peace Prize", "Grammy Award"*

Exploration Optimization

The exploration order has a high impact on the efficiency of graph exploration. In TriAD, we determine the best exploration order by using stored summary statistics and employ a bottom-up dynamic programming algorithm which minimizes the costs of the exploration order. At each DP step, we calculate the cost of the partial plan considered so far and prune whenever the current branch cannot contribute to the plan with the least cost anymore. The cost of an entire exploration plan that is represented by a fixed order of triple patterns $R_1, \ldots, R_n$ can thus be estimated as follows.

$$Cost(\langle R_1, \ldots, R_n \rangle) \propto Card(R_1) + \sum_{i=2}^{n} \left(Card(R_i) \prod_{j=1}^{i} Sel(R_i, R_j) \right) \tag{2}$$

Here, $Card(R_i)$ denotes the cardinality of query pattern R_i, and $Sel(R_i, R_j)$ is the join-selectivity of two patterns R_i, R_j, i.e., the ratio of the number of triples joined over the size of the cross product between the triples in R_i and R_j.

5. EVALUATION

For an evaluation of the pruning effectiveness of the summary graph, we focus on a centralized setup of TriAD, and we compare to four state-of-the-art RDF engines, RDF-3X [7], BitMat [2], MonetDB [11] and Trinity.RDF [14] (the latter being deployed on a comparable centralized setting). For a detailed evaluation of TriAD under a distributed setting, we refer to [3].

We used the widely popular LUBM[4] benchmark (in N3 format) and generated the data using UBA 1.7[5]. Concerning queries, we used the benchmark queries published in [2] which are also used by Trinity.RDF. For constructing the summary graph, we employ METIS 5.1[6] as our graph partitioner. To achieve a better performance during partitioning, we ignored edges connecting string literals, resulting in both space and time savings.

[4] http://swat.cse.lehigh.edu/projects/lubm/
[5] http://www.l3s.de/~minack/rdf-fulltext-benchmark/
[6] http://glaros.dtc.umn.edu/gkhome/metis/metis/download

	TriAD	TriAD-SG (17K)		Trinity	RDF-3X		MonetDB		BitMat	
		Full GE	1-hop GE	RJ	.RDF	(cold)	(warm)	(cold)	(warm)	(cold)	(warm)
Q1	427	**97**	412	382	281	38,802	27,702	10,600	1,500	1,078	1,053
Q2	**117**	140	139	137	132	32,936	347	279	174	3,055	3,030
Q3	210	**31**	75	321	110	27,692	27,678	10,900	1,700	47	40
Q4	2	**1**	3.1	2.3	5	76	2	39	25	5,421	5,357
Q5	0.5	**0.2**	0.8	0.9	4	1	1	80	23	6	5.8
Q6	19	**1.8**	2.5	3	9	59	7	130	51	132	128
Q7	693	711	712	1447	**630**	35,485	1,086	10,100	1,700	1,642	1,583
Geo.-Mean	39	**14**	29	40	46	1,280	170	748	216	277	362

Table 1: LUBM-160 – Query processing times (in *ms*)

Results. Table 1 shows the runtimes of TriAD and TriAD-SG (TriAD with the summary graph pruning enabled) against the competing engines over the LUBM 160 dataset with about 27 million triples. To better interpret the results, we categorized the queries Q1–Q7 into 3 groups – non-selective (Q2), selective in both input and output size (Q4,Q5,Q6), and selective only in the output size (Q1,Q3,Q7). We can observe that for the selective queries Q4,Q5,Q6, the advantage of join-ahead pruning by the summary graph (TriAD-SG - Full GE) boosts the performance of TriAD. For queries Q1,Q3,Q7, which are selective only in their final output size, TriAD generates large intermediate results thus being slower than Trinity.RDF. This issue is addressed in TriAD-SG (Full GE) via join-ahead pruning and the full graph exploration. For query Q2, which is a non-selective, single join query, the benefits of the summary graph are however not noticeable, and with its additional overhead TriAD-SG actually becomes slower than TriAD.

Summary size	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Geo-.Mean
10K	153	141	30	0.8	0.5	1.5	692	16
17K	97	140	31	0.7	0.2	1.8	711	14
20K	86	140	36	0.9	0.5	1.5	702	16

Table 2: Impact of summary graph partitions for LUBM-160

Impact of Summary Graph Sizes

The number of summary graph partitions directly affects the performance of the system as highlighted in Table 2. With a smaller number of partitions, each supernode comprises of many triples. Thus, even though the join-ahead pruning can be done quickly over smaller summary graphs, due to large supernode sizes, the overall number of pruned tuples remains low, thus making the second-stage query processing considerably more expensive. Thus, the right choice of the summary graph size has a crucial impact on the overall performance.

Approach	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Geo-.Mean
Full GE	22	3	22	0.03	0.003	0.5	82	1.3
1-hop GE	13	4	13	1.4	0.3	1.8	17	3.9
RJ	312	6	312	1.4	0.4	1.6	767	16.3

Table 3: 1-hop and full graph exploration (GE) vs. relational joins (RJ) for LUBM-160

Graph Exploration vs. Relational Joins

Finally, we compared three approaches for processing the summary graph: (1) full graph exploration (Full GE), (2) 1-hop graph exploration (1-hop GE), (3) a conventional form of relational joins (RJ). Table 3 shows the runtime performance of the three approaches over a summary graph with 17K partitions for the LUBM 160 dataset. We can clearly observe that, in a relational approach, there is a penalty incurred for generating large intermediate relations. This is avoided entirely in graph exploration (Full GE) without increasing the number of false-positive bindings. On the other hand, as expected, the 1-hop exploration performs faster than the full exploration (Full GE) for the complex queries Q1,Q3,Q7, but it also retains a lot of false positives, which in turn makes the second stage of processing more costly (Table 1).

6. CONCLUSIONS

In this paper, we presented a detailed discussion of graph summarization for the purpose of join-ahead pruning in large RDF graphs. Specifically, we discussed details of generating and indexing the summary graph, and how to leverage the summary graph for effective join-ahead pruning. We build on the TriAD framework to demonstrate the effect of join-ahead pruning via a locality-based form of graph summarization. In both, the centralized and distributed settings of TriAD, we implemented the summary graph at the master node and used a two-staged query processing strategy to achieve a better overall query processing performance. In future work, we plan to compare different summarization approaches based on query workloads and bisimulation, and to compare its effectiveness with this still rather simple (i.e., locality-based) form of summarization.

7. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. SW-Store: A vertically partitioned DBMS for Semantic Web data management. *The VLDB Journal*, 18(2), 2009.

[2] 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*, 2010.

[3] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: A Distributed Shared-Nothing RDF Engine Based on Asynchronous Message Passing. *SIGMOD*, 2014.

[4] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11), 2011.

[5] G. Karypis and V. Kumar. Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.

[6] T. Milo and D. Suciu. Index Structures for Path Expressions. In *ICDT*. 1999.

[7] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1), 2010.

[8] T. Neumann and G. Weikum. x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases. *PVLDB*, 2010.

[9] F. Picalausa, Y. Luo, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. A structural approach to indexing triples. In *ESWC*, 2012.

[10] K. Rohloff and R. E. Schantz. Clause-iteration with MapReduce to scalably query datagraphs in SHARD graph-store. In *DIDC*, 2011.

[11] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2), 2008.

[12] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1), 2008.

[13] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. TripleBit: a Fast and Compact System for Large Scale RDF Data. *PVLDB*, 6(7), 2013.

[14] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *PVLDB*, 6(4), 2013.

[15] X. Zhang, L. Chen, Y. Tong, and M. Wang. EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In *ICDE*, 2013.

[16] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering SPARQL Queries via Subgraph Matching. *PVLDB*, 4(8), 2011.