A Fully Parallel Framework for Analyzing RDF Data

Long Cheng[1], Spyros Kotoulas[2], Tomas E Ward[3], Georgios Theodoropoulos[4]

[1] Technische Universität Dresden, Germany [2] IBM Research, Ireland
[3] National University of Ireland Maynooth, Ireland [4] Durham University, UK
long.cheng@tu-dresden.de, spyros.kotoulas@ie.ibm.com
tomas.ward@nuim.ie, theogeorgios@gmail.com

Abstract. We introduce the design of a fully parallel framework for quickly analyzing large-scale RDF data over distributed architectures. We present three core operations of this framework: dictionary encoding, parallel joins and indexing processing. Preliminary experimental results on a commodity cluster show that we can load large RDF data very fast while remaining within an interactive range for query processing.

1 Introduction

Fast loading speed and query interactivity are important for exploration and analysis of RDF data at Web scale. In such scenarios, large computational resources would be tapped in a short time, which requires very fast data loading of the target dataset(s). In turn, to shorten the data processing life-cycle for each query, exploration and analysis should be done in an interactive manner. In the context of these conditions, we follow the following design paradigm.

Model. We employ the commonly used relational model. Namely, RDF data is stored in the form of triples and SPARQL queries are implemented by a sequence of lookups and joins. We do not use the graph-based approaches, because they focus on subgraph matching, which is not suitable for handling large-scale RDF data, as described in [1]. Moreover, for a row-oriented output format, graph exploration is not sufficient to generate the final join results, as presented in [2] and graph-partitioning approaches are too costly, in terms of loading speed.

Parallelism. We parallelize all operations such as dictionary encoding, indexing and query processing. Although asynchronous parallel operations (such as joins) have been seen to improve load balancing in state-of-art systems [2], we still adopt the conventional synchronous manner, since asynchronous operations always rely on specified communication protocols (*e.g.* MPI). To remedy the consequent load-imbalance problem, we focus on techniques to improve the implementations of each operation. For example, for a series of parallel joins, we keep each join operation load-balanced.

Performance. We are interested in the performance in both data loading and querying. In fact, current distributed RDF systems generally operate on a trade-off between loading complexity and query efficiency. For example, the similar-size partitioning method [3, 4] offers superior loading performance at the cost of more complex/slower querying, and the graph-based partitioning approach [2, 5] requires significant computational effort for data loading and/or partitioning. Given the trade-offs between the two

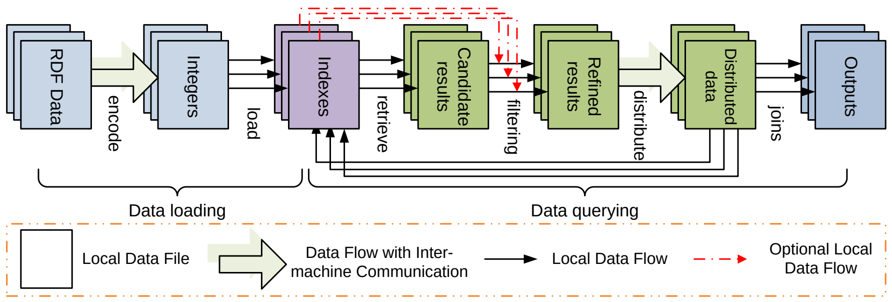

Fig. 1. General design of our parallel framework.

approaches, we combine elements of both to achieve fast loading while still keeping query time in an interactive range.

Our parallel framework is shown in Figure 1. The entire data process is divided into two parts: data loading and data querying. (1) The equal-size partitioned raw RDF data at each computation node (core) is encoded in parallel in the form of integers and then loaded in memory in local indexes (without redistributing data). (2) Based on the query execution plan, the candidate results are retrieved from the built indexes, and parallel joins are applied to formulate the final outputs. In the latter process, local filters[1] at each node can be used to reduce/remove the retrieved results that have no contribution for the final outputs, and the redistributed data during *parallel joins* can be used to create additional sharded indexes.

2 Core Operations

Triple Encoding. We utilise a distributed dictionary encoding method, as described in [6,7], to transform RDF terms into 64-bit integers and to represent statements (aligned in memory) using this encoding. Using a straightforward technique and an efficient skew-handling strategy, our implementation [6] is shown to be notable faster than [8] and additionally supports *small incremental updates*.

Parallel Joins. Based on existing indexes, we can lookup the candidate results for each graph pattern and then use joins to compute SPARQL queries. For the most critical join operation, *parallel hash joins* [3] are commonly used in current RDF systems. However, they always bring in load-imbalance problems. The reason is that the terms in real-world Linked Data are highly skewed [9]. In comparison to that, our implementation adopts the *query-based distributed joins* we proposed in [10–13] so as to achieve more efficient and robust performance on each join operation in the presence of different query workloads.

Two-tier Indexing. We adopt an efficient two-tier index architecture we presented in [14]. We build the primary index l_1 for the encoded triples at each node using a modified *vertical partitioning* approach [15]. Different from [15], to speedup the load process, we do not do any sort operation, but just insert each tuple in a corresponding *vertical table*. For join operations, we could have to redistribute a large number of

[1] Though our system supports filtering operations, we do not give the details in this paper.

(intermediate) results around all computation nodes, which is normally very costly. To remedy this, we employ a bottom-up dynamic programming-like parallel algorithm to build a multi-level secondary index (l_2 ... l_n), based on each query execution plan. With that, we will simply *copy* the redistributed data of each *join* to the local secondary indexes, and these parts of data will be re-used by other queries that contain patterns in common, so as to reduce (or remove) the corresponding network communication during the execution. In fact, according to the terminology regarding *graph partitioning* used in [5], the k-level index l_k on each node in our approach will dynamically construct a k-hop subgraph. This means that our method essentially does dynamic graph-based partitioning based on the query load, starting from an initial equal-size partitioning. Therefore, our approach can combine the loading speed of similar-size partitioning with the execution speed of graph-based partitioning in an analytical environment.

3 Preliminary Results

Experiments were conducted on 16 IBM iDataPlex® nodes with two 6-core Intel Xeon® X5679 processors, 128GB of RAM and a single 1TB SATA hard-drive, connected using Gigabit Ethernet. We use Linux kernel version 2.6.32-220 and implement our method using X10 version 2.3, compiled to C++ with gcc version 4.4.6.

Data Loading. We test the performance of *triple encoding* and *primary index building* through loading 1.1 billion triples (LUBM8000 with indexes on P, PO and PS) in memory. The entire process takes 340 secs, for an average throughput of 540MB or 3.24M triples per second (254 secs to encode triples and 86 secs to build the primary index).

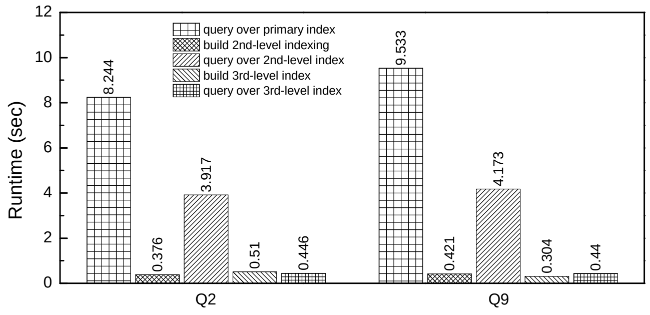

Fig. 2. Runtime over different indexes using 192 cores.

Data Querying[2]. We implement queries over the indexes l_1, l_2 and l_3 to examine the efficiency of our *secondary indexes*. We run the two most complex queries Q2 and Q9 of LUBM. As we do not support RDF inference, the query Q9 is modified as below so as to guarantee that we can get results for each basic graph pattern.

Q9: select ?x ?y ?z where { *?x rdf:type ub:GraduateStudent. ?y rdf:type ub:FullProfessor. ?z rdf:type ub:Course. ?x ub:advisor ?y. ?y ub:teacherOf ?z. ?x ub:takesCourse ?z.* }

[2] The results presented here is a mirror of our previous work [14].

To focus on analyzing the core performance only, we report times for the operations of *results retrieval* and the *joins* (namely we are excluding the time to decode the output) in the execution phase.

The results in Figure 2 show that the secondary indexes can obviously improve the query performance. Moreover, the higher the level of index is, the lower the execution time. Additionally, it can be seen that building a high-level index is very fast, taking only hundreds of *ms*, which is extremely small compared to the query execution time.

We did not employ the *query-based* joins as mentioned in our query execution presented here, as we found the data skew in our tests was not obvious (due to the nature structure of the LUBM benchmark). We plan to integrate the joins with the development of our system, and then present more detailed results using much more complex workloads (*e.g.*, similar to the one used in [4]).

Acknowledgments. This work was supported by the DFG in grant KR 4381/1-1. We thank Markus Krötzsch for comments that greatly improved the manuscript.

References

1. Sun, Z., Wang, H., Wang, H., Shao, B., Li, J.: Efficient subgraph matching on billion node graphs. Proc. VLDB Endow. **5**(9) (May 2012) 788–799
2. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing. In: SIGMOD. (2014) 289–300
3. Weaver, J., Williams, G.T.: Scalable RDF query processing on clusters and supercomputers. In: SSWS. (2009)
4. Kotoulas, S., Urbani, J., Boncz, P., Mika, P.: Robust runtime optimization and skew-resistant execution of analytical SPARQL queries on PIG. In: ISWC. (2012) 247–262
5. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL querying of large RDF graphs. Proc. VLDB Endow. **4**(11) (2011) 1123–1134
6. Cheng, L., Malik, A., Kotoulas, S., Ward, T.E., Theodoropoulos, G.: Efficient parallel dictionary encoding for RDF data. In: WebDB. (2014)
7. Cheng, L., Malik, A., Kotoulas, S., Ward, T.E., Theodoropoulos, G.: Scalable RDF data compression using X10. arXiv preprint arXiv:1403.2404 (2014)
8. Urbani, J., Maassen, J., Drost, N., Seinstra, F., Bal, H.: Scalable RDF data compression with MapReduce. Concurrency and Computation: Practice and Experience **25**(1) (2013) 24–39
9. Kotoulas, S., Oren, E., Van Harmelen, F.: Mind the data skew: Distributed inferencing by speeddating in elastic regions. In: WWW. (2010) 531–540
10. Cheng, L., Kotoulas, S., Ward, T.E., Theodoropoulos, G.: QbDJ: A novel framework for handling skew in parallel join processing on distributed memory. In: HPCC. (2013) 1519–1527
11. Cheng, L., Kotoulas, S., Ward, T.E., Theodoropoulos, G.: Efficiently handling skew in outer joins on distributed systems. In: CCGrid. (2014) 295–304
12. Cheng, L., Kotoulas, S., Ward, T.E., Theodoropoulos, G.: Robust and efficient large-large table outer joins on distributed infrastructures. In: Euro-Par. (2014) 258–269
13. Cheng, L., Kotoulas, S., Ward, T.E., Theodoropoulos, G.: Robust and skew-resistant parallel joins in shared-nothing systems. In: CIKM. (2014)
14. Cheng, L., Kotoulas, S., Ward, T.E., Theodoropoulos, G.: A two-tier index architecture for fast processing large RDF data over distributed memory. In: HT. (2014) 300–302
15. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: VLDB. (2007) 411–422