A Two-tier Index Architecture for Fast Processing Large RDF Data over Distributed Memory

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

[1] National University of Ireland Maynooth, Ireland [2] IBM Research, Ireland
[3] Technische Universität Dresden, Germany [4] Durham University, UK

long.cheng@tu-dresden.de, spyros.kotoulas@ie.ibm.com, tomas.ward@nuim.ie, theogeorgios@gmail.com

ABSTRACT

We propose an efficient method for fast processing large RDF data over distributed memory. Our approach adopts a two-tier index architecture on each computation node: (1) a light-weight primary index, to keep loading times low, and (2) a dynamic, multi-level secondary index, calculated as a by-product of query execution, to decrease or remove inter-machine data movement for subsequent queries that contain the same graph patterns. Experimental results on a commodity cluster show that we can load large RDF data very quickly in memory while remaining within an interactive range for query processing with the secondary index.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Distributed Databases, Query Processing

Keywords

Distributed RDF Processing; Dynamic Indexing

1. INTRODUCTION

Responding to the rapid growth of Linked Data, several approaches for distributed RDF data processing have been proposed [18, 16, 10, 15], along with clustered versions of more traditional approaches [9, 2, 17]. Depending on the data partitioning and placement patterns, these solutions can be divided into four categories: (1) *Similar-size partitioning*: Partitions containing similar volumes of raw triples are placed on each computation node without a global index. During query processing, nodes provide bindings for each triple pattern and formulate the intermediate (or final) results using *parallel joins* [18, 15]. (2) *Hash-based partitioning*: Exploiting the fact that SPARQL queries often contain "star" graph patterns, triples under this scheme are commonly hash partitioned (by subject) across multiple machines and accessed in parallel at query time [16, 11]. (3) *Sharded/Partitioned indexes*: Perhaps the approach closest to centralized stores, triple indexes in the form of SPO, OPS etc are distributed across the nodes in a cluster and stored as a B-Tree [9, 17]. (4) *Graph-based partitioning*: Graph partitioning algorithms are used to partition RDF data in a manner that triples close to each other can be assigned to the same computation node. SPARQL queries generally take the form of graph pattern matching so that sub-graphs on each computation node can be matched independently and in parallel, as much as possible [10].

In general, the techniques outlined above operate on a trade-off between loading complexity and query efficiency, with the earlier ones in the list offering superior loading performance at the cost of more complex/slower querying and the latter ones requiring significant computational effort for loading and/or partitioning. In this paper, we are proposing an efficient parallel way that *combine the loading speed of similar-size partitioning with the execution speed of graph-based partitioning*.

2. OUR APPROACH

The main elements of our approach are: (1) We maintain a local light-weight primary index supporting very fast data loading and retrieval. (2) Secondary indexes supporting non-trivial access patterns are built dynamically, as a byproduct of query execution. In the following, we refer to the primary index as (l_1) and secondary indexes as 2nd-level (l_2), 3rd-level (l_3), etc.

Triple Encoding. We first transform RDF terms into 64-bit integers and represent statements using this encoding. We utilise a distributed dictionary encoding method as described in our previous work [8]. The overall implementation strategy for each node and the corresponding data flow are shown in Figure 1.

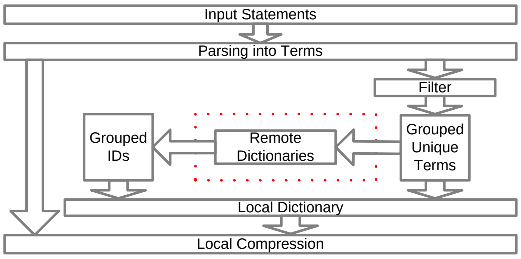

Figure 1: Workflow of triple encoding at each node.

Every statement in the input set is parsed and split into individual *terms*, namely, *subject*, *predicate*, and *object*. Duplicates are locally eliminated, and the extracted set of *unique* terms is then divided into individual groups according to their hash values. The groups of unique terms are then pushed to the responsible remote dictionaries for encoding. After that, every node builds a local dictionary, for encoding the parsed statements, based on the grouped

HT'14, September 1–4, 2014, Santiago, Chile.
ACM 978-1-4503-2954-5/14/09.
http://dx.doi.org/10.1145/2631775.2631789.

unique terms and the corresponding group of ids received from remote nodes.

Primary Index. After encoding, we build the primary index l_1 for the encoded triples at each node. We use a modified *vertical partitioning* approach [1] to decompose the local data into multiple parts. Triples in [1] are placed into n two-column *vertical tables* (n is number of unique properties), and all the *subjects* in each table are sorted. In comparison, we only insert each tuple in an **unordered** list in a corresponding *vertical table*. To support multiple access patterns, we build additional tables. By default, we build $P \rightarrow SO$, $PS \rightarrow O$ and $PO \rightarrow S$, corresponding to the most common access patterns. Note that there is no communication over the network for this step.

Parallel Hash Joins. Once we have built the primary index, we can compute SPARQL queries through a sequence of lookups and joins. For a basic graph pattern (BGP), looking up the results can be implemented in parallel and independently for each node. Regardless, a *join* between any two sub-queries can not be executed independently since we have no guarantee that join keys will be located on the same node. We adopt the parallel hash-join implementation here, namely, results of each subquery are redistributed among computation nodes by hashing the values of their join keys, so as to ensure that the appropriate results for the join are co-located [18].

Secondary Indexes. For join operations, as we have to redistribute all results for each triple pattern as well as the intermediate results, data transfers between each node become costly. To remedy this shortcoming, we employ a bottom-up dynamical programming parallel algorithm to build secondary indexes (l_2 ... l_n), based on each query execution plan.

For simplification, here, we just give a simple example to show the process of building the 2nd-level index l_2 based on a join between two basic graph patterns. As shown in Algorithm 1, the first three steps (lines 1-3) is actually a *parallel hash joins* processing. Regardless, after that, the redistributed results will be kept locally in l_2, according to the non-variables appearing in the responsible BGP. For instance, the redistributed results of the BGP <?s p1 ?o> will be added into the vertical table $p1 \rightarrow SO$ of l_2.

Algorithm 1 Implementation of building l_2 at each node

```
Phase 1: Tuple redistribution
1: retrieve result $r_i$ ($i = 1, 2$) of each BGP from the index $l_1$
2: redistribute $r_i$ to all nodes according to hash values of join keys
Phase 2: $l_2$ index building
3: implement local joins and formulate outputs
4: insert received tuples $r'_i$ into local $l_2$
```

Since the index is constructed by a simple *copy* of the redistributed data, which is introduced by a *join* of a query, the secondary indexes can be re-used by other queries that contain patterns in common. In fact, according to the terminology regarding *graph partitioning* used in [10], the 2nd-level index on each node will construct a 2-hop subgraph, the 3rd-level one will be a 3-hop subgraph, and l_k will become to k-hop subgraph. This means that our method essentially does dynamic graph-based partitioning starting from an initial equal-size partitioning, based on the query load. Therefore, our approach can combine their advantages on fast data loading and efficient querying.

3. EVALUATION

Platform. We use 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 [3] version 2.3, compiled to C++ with gcc version 4.4.6.

Setup. We load LUBM(8000), containing about 1.1 billion triples, and run the two most complex queries Q2 and Q9. As we do not support RDF inference, we use a modified version shown in the Appendix. To focus on analyzing the core performance only, we do not count the time spent on *parsing*, *planning*, *dictionary lookup* or *result output* as described in [4].

Data Loading. We load 1.1 billion triples and build three primary indexes (on P, PO and PS) in memory. As shown in Table 1, it takes 254 seconds to encode triples and 86 seconds to build the primary index l_1, for an average throughput of 540MB or 3.24M triples per second. This is faster than any other implementation in the literature.

Table 1: Time to load 1.1 billion triples using 192 cores

Triple encoding:	254 seconds
Building l_1 (P, PO, PS):	86 seconds
Total:	**340 seconds**

Data Querying. We examine the runtime of Q2 and Q9 using l_1, l_2 and l_3. Meanwhile, we also record the time cost to build indexes. Figure 2 shows that the secondary index can obviously improve the query performance. Moreover, the higher the level of index is, the lower the execution time. At the same time, we can also see that the operation of building a high-level index is very fast, taking only hundreds of *ms*, which is extremely small compared to the query execution time.

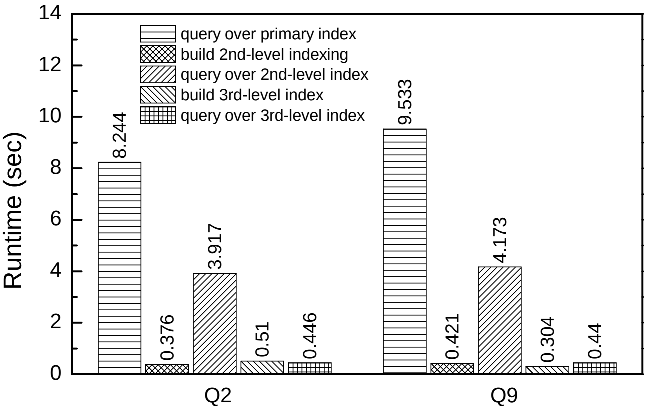

Figure 2: Runtime over different indexes using 192 cores.

4. CONCLUSION

In this work, we propose a dynamic two-tier index architecture designed for fast processing large RDF data over distributed memory. Our experimental results demonstrate that the approach can both load and query large RDF datasets quickly.

We will investigate extensions to our design through the application of methods for *skew handling* [15, 5, 6, 7], *index size reduction* [14] and *incremental sorting* [12, 13] which should further improve performance. Our long term goal is to develop a highly scalable distributed analysis framework for extreme-scale RDF data.

Acknowledgments. This work is supported by the Irish Research Council and IBM Research Ireland.

5. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very large Data Bases*, VLDB' 07, pages 411–422, 2007.

[2] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov. OWLIM: A family of scalable semantic repositories. *Semantic Web*, 2(1):33–42, 2011.

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices*, 40(10):519–538, 2005.

[4] L. Cheng, S. Kotoulas, T. Ward, and G. Theodoropoulos. Runtime characterization of triple stores. In *Proceedings of the 15th IEEE International Conference on Computational Science and Engineering*, CSE' 12, pages 66–73, 2012.

[5] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos. QbDJ: A novel framework for handling skew in parallel join processing on distributed memory. In *Proceedings of the 15th IEEE International Conference on High Performance Computing and Communications*, HPCC' 13, pages 1519–1527, 2013.

[6] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos. Efficient handling skew in outer joins on distributed systems. In *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid' 14, pages 295–304, 2014.

[7] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos. Robust and efficient large-large table outer joins on distributed infrastructures. In *Proceedings of the 20th European Conference on Parallel Processing*, Euro-Par' 14, pages 258–269, 2014.

[8] L. Cheng, A. Malik, S. Kotoulas, T. E. Ward, and G. Theodoropoulos. Efficient parallel dictionary encoding for RDF data. In *Proceedings of the 17th International Workshop on the Web and Databases*, WebDB' 14, 2014.

[9] O. Erling and I. Mikhailov. Virtuoso: RDF support in a native RDBMS. In *Semantic Web Information Management*, pages 501–519. Springer, 2010.

[10] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011.

[11] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. M. Thuraisingham. Heuristics-based query processing for large RDF graphs using cloud computing. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1312–1327, 2011.

[12] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[13] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 297–308, 2009.

[14] K. Kim, B. Moon, and H.-J. Kim. R3F: RDF triple filtering method for efficient SPARQL query processing. *World Wide Web*, pages 1–41, 2013.

[15] S. Kotoulas, J. Urbani, P. Boncz, and P. Mika. Robust runtime optimization and skew-resistant execution of analytical SPARQL queries on PIG. In *Proceedings of the 11th International Semantic Web Conference*, ISWC' 12, pages 247–262. 2012.

[16] K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the MapReduce software framework: The SHARD triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, 2010.

[17] B. Thompson and M. Personick. Bigdata: The semantic web on an open source cloud. In *International Semantic Web Conference*, 2009.

[18] J. Weaver and G. T. Williams. Scalable RDF query processing on clusters and supercomputers. In *The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, SSWS' 09, 2009.

APPENDIX

The rewritten LUBM SPARQL queries Q2 and Q9 used in our evaluation are as follows.

Q2: select ?x ?y ?z where { ?x rdf:type lubm:GraduateStudent. ?y rdf:type lubm:Department. ?z rdf:type lubm:University. ?y lubm:sub OrganizationOf ?z. ?x lubm:memberOf ?y. ?x lubm:undergraduate DegreeFrom ?z.}

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.}