Exploiting Wide Property Tables Empowered by Inverse Properties for Efficient Distributed SPARQL Query Evaluation

Guilherme Schievelbein, Victor Anthony Arrascue Ayala, Fang Wei-Kleiner and Georg Lausen

University of Freiburg, Georges-Köhler Allee, Geb. 51, 79110 Freiburg, Germany
{schieveg,arrascue,fwei,lausen}@informatik.uni-freiburg.de
http://dbis.informatik.uni-freiburg.de

Abstract. Translating SPARQL to Spark SQL has been proposed to achieve better scalability in query evaluation. Recent investigations show that the database design for storing the RDF-graph plays a significant role in the performance, due to intrinsic characteristics of Spark's computation model. The analysis points to the interesting fact that a Wide Property Table (WPT), a single-table design with one row for each subject and one column for each property, has very nice properties for storing RDF-graphs. In addition to WPT's simplicity, SPARQL queries, in particular those with many joins on subjects, are translated to an efficient Spark execution plan. We aim to extend the WPT with inverse properties to broaden this benefit to other kinds of queries. Thus, in this paper we propose a framework which can leverage one or a combination of WPTs extensions. Our experiments on a widely used benchmark reveal that a combination of three different kinds of WPT together leads to the best performance for almost all query types but the linear-shaped ones.

1 Introduction and Related Work

Translation from SPARQL to Spark SQL is widely researched because it makes it possible to benefit from the robustness and scalability of cloud computing infrastructure. In Spark, a database design involves a collection of DataSets, which are at the physical level divided into partitions (set of rows), and distributed and replicated among cluster nodes. An example of such a design is the Wide Property Table (WPT), a very sparse single-table design with one row for each subject and as many columns as the number of distinct properties. Some notable examples of systems on top of Spark are: S2RDF [7] and SPARQLGX [5] which use another design based on vertically partitioning by predicate (VP), PRoST [4] and the approach by Hassan et al. [6] which combine VP and WPT, the second even multiple subsets of WPTs. As pointed out by a recent analysis by Arrascue et al. [2], the superior performance of the WPT design is due to its favorable

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2456/paper21.pdf

characteristics: 1) since a WPT partition has all information related to a set of subjects, star-shaped queries with many on-subject joins are not translated to Spark SQL joins, which might eventually require shuffling. Instead, they are translated to operations which can be solved locally and result in a reduced number of stages in the execution plan; 2) the number of partitions is large enough to take full advantage of the parallelism. Thus, we aim to extend the WPT with inverse properties so that even more graph pattern bindings can locally occur within a single partition. Our proposed framework makes it possible to leverage multiple of these WPT extensions. This allowed us to carry out a performance evaluation and find out how to best make use of WPTs with inverse properties.

2 Approach

We propose a framework capable of using multiple WPT extensions from the same RDF-graph to evaluate SPARQL queries. We achieve this by generating an intermediate abstract tree of operation nodes to represent a given SPARQL query. The central criterion to build it is the minimization of the number of join operations. In this tree each node can independently use an available WPT design. The leaves contain a graph pattern (a single or conjunction of triple patterns), that can be evaluated without a Spark SQL join operation in a given database. Intermediate nodes represent join operations on the common variables of their children nodes. Thus, to minimize the number of join operations necessary for the query evaluation the algorithm takes all available designs given as input into account. Given that WPT works well for on-subject joins, we consider here also its object counterpart, the Inverse Wide Property Table (iWPT). An iWPT contains one distinct column for each property of the RDF-graph and each row contains the information about an object. Therefore, graph patterns with a common object resource can be processed with a single scan of the iWPT. These two tables, WPT and iWPT, can be joined to have in a single row all information connected one-hop away from a resource, regardless of the predicate direction. Therefore, we consider here two variations, one obtained through a full outer join (jWPT-outer) and an inner join (jWPT-inner) between the WPT and iWPT. The schema of the WPT, iWPT, and jWPT, given an RDF-graph with n distinct properties, is shown in Table 1.

WPT			
s	p_0	...	p_n

iWPT			
p_n^{-1}	...	p_0^{-1}	**o**

jWPT						
p_n^{-1}	...	p_0^{-1}	**r**	p_0	...	p_n

Table 1: Schema of a WPT, iWPT, and jWPT.

An example of how the abstract tree is generated on top of some of these WPT extensions is provided in Figure 1. Input to our algorithm is the SPARQL query (A) which results in the graph pattern illustrated in (B), and the set of database designs where the operations can be executed. As the figure shows, when only WPT is available (C) all on-subject patterns are grouped together, while others are executed independently on the same table. When also the iWPT is available, the same occurs with on-object joins, which are now grouped in one

leaf (D). Finally, the complete graph pattern is placed in a single leaf when the jWPT is available (E).

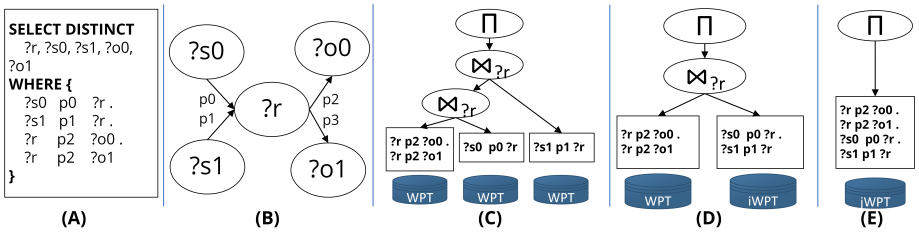

Fig. 1: (A) SPARQL query; (B) Resulting graph pattern; (C-E) Abstract-trees when only WPT (C), WPT and iWPT (D), and WPT, iWPT and jWPT (E) (is) are available

3 Evaluation results

We performed our tests on a small cluster of 10 machines, 1 master and 9 workers, connected via Gigabit Ethernet connection. Each machine is equipped with 32GB of memory, 4TB of disk space and with a 6 Core Intel Xeon E5-2420 processor. The cluster runs Cloudera CDH 5.10.0 with Spark 2.2 on Ubuntu 14.04. Yarn, the resource manager, in total uses 198 GB and 108 virtual cores. The tests were run with using a WatDiv dataset [1] with around 100M triples. In Table 2 we show the approximate size and number of rows of the created tables. The evaluation results are displayed in the graphs from Figure 2. We compare the execution times of the queries using 4 different combinations of enabled data models: WPT only (C1), WPT and iWPT (C2), jWPT-outer only (C3), and WPT, iWPT and jWPT-inner (C4). We tried other combinations, such as using VP with WPT variations, but we report only the combinations that led to the best performance. The query set is evaluated 25 times in random orders. We prune execution time outliers using the interquartile range (IQR) to set an upper fence. Finally, the results are aggregated by the query shapes available in Watdiv: complex (Wat-C), snowflake (Wat-F), linear (Wat-L), and star (Wat-S).

The graph (A) in Figure 2 shows the average time when considering the entire Watdiv benchmark query set. Since Watdiv is strongly biased towards queries which only contain on-subject joins, we show in the graph (B) the performance evaluation on a reduced Watdiv query set whose queries contain at least two joins, one on a subject and one on an object, and excluding completely linear-shaped ones. This allows us to closely analyse the benefit of jWPT, which reduces the number of Spark SQL join operations the most. We can observe from the graphs that the combination of WPT and iWPT in C2 performs slightly worse than WPT only (C1), except for complex queries, but the difference is not very significant. In the case of C3, jWPT-outer only, when considering the reduced query set, there is an improvement over C1 and C2 for all query shapes but

the complex ones. Interestingly, the same does not occur when considering the full query set. This can be attributed to the higher number of rows present in the jWPT-outer. Finally, C4, the combination of WPT, iWPT and jWPT-inner, resulted in the best performance for all query shapes, but the linear-shaped ones, showing that further improvements can be made for the efficient evaluation of this specific type of query.

Data model	Size	Tuples
WPT	1.1GB	5.2M
iWPT	1.2GB	9.7M
jWPT-outer	2.3GB	10.2M
jWPT-inner	1.8GB	4.7M

Table 2: Tables statistics.

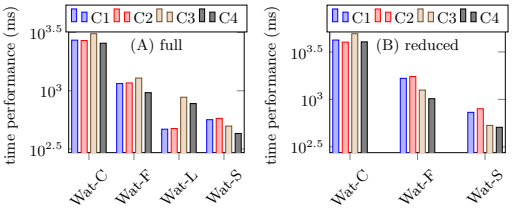

Fig. 2: Performance of (A)full and (B)reduced query sets

4 Conclusions and Future Work

In this paper we show the benefit of extending WPTs with inverse properties. Similar ideas can be found in a commercial system [3], which however are not suitable for a computer cluster. Our experiments not only demonstrate the applicability of our approach to leverage multiple designs from the same RDF-graph, but it made it possible to extend the performance analysis to combinations which were never tried before. The results show that the combination of three different designs WPT, iWPT, jWPT-inner (C4) leads to the best performance for most query types, while there is still room for improvement to process linear ones. This indicates that when dealing with multiple designs there exists a trade-off between the number of joins, and the number of rows in all involved tables.

References

1. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: ISWC (2014)
2. Ayala, V.A.A., Koleva, P., Alzogbi, A., et al.: Relational schemata for distributed SPARQL query processing. In: SBD@SIGMOD (2019)
3. Bornea, M.A., Dolby, J., Kementsietsidis, A., et al.: Building an efficient RDF store over a relational database. In: SIGMOD (2013)
4. Cossu, M., Färber, M., Lausen, G.: Prost: Distributed execution of SPARQL queries using mixed partitioning strategies. In: EDBT (2018)
5. Graux, D., Jachiet, L., Genevès, P., Layaïda, N.: SPARQLGX: efficient distributed evaluation of SPARQL with apache spark. In: ISWC (2016)
6. Hassan, M., Bansal, S.K.: Data partitioning scheme for efficient distributed RDF querying using apache spark. In: ICSC (2019)
7. Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF querying with SPARQL on spark. PVLDB (2016)