SPARQL Graph Pattern Processing with Apache Spark

Hubert Naacke, Bernd Amann
Sorbonne Universités, UPMC Univ Paris 06, LIP6 UMR 7606, Paris, France
firstName.lastName@lip6.fr

Olivier Curé
Université Paris-Est Marne la Vallée, France
olivier.cure@u-pem.fr

ABSTRACT

A common way to achieve scalability for processing SPARQL queries is to choose MapReduce frameworks like Hadoop or Spark. Processing basic graph pattern (BGP) expressions generating large join plans over distributed data partitions is a major challenge in these frameworks. In this article, we study the use of two distributed join algorithms, partitioned join and broadcast join, for the evaluation of BGP expressions on top of Apache Spark. We compare five possible implementation and illustrate the importance of cautiously choosing the physical data storage layer and of the possibility to use both join algorithms to efficiently take account of existing data partitioning schemes. Our experimentations with different SPARQL benchmarks over real-world and synthetic workloads emphasize that hybrid join plans introduce more flexibility and often achieve better performance than single kind join plans.

ACM Reference format:
Hubert Naacke, Bernd Amann and Olivier Curé. 2017. SPARQL Graph Pattern Processing with Apache Spark. In *Proceedings of GRADES'17, Chicago, IL, USA, May 19, 2017,* 7 pages.
DOI: http://dx.doi.org/10.1145/3078447.3078448

1 INTRODUCTION

The features expected from modern Resource Description Framework (RDF) stores are reminiscent of many other big data applications which, instead of reimplementing physical distributed data processing layers from scratch, prefer to rely on MapReduce frameworks, *e.g.*, Apache's Hadoop[13] or Spark [14] or other scalable NoSQL data stores [10]. MapReduce frameworks are built on horizontally scalable shared nothing architectures to achieve data parallelism and efficient parallel processing of large data sets. However, this horizontal scalability comes at the cost of complicating the efficient evaluation of distributed join plans which might generate important data transfer costs between cluster nodes. A common goal of existing solutions is to achieve data-to-query locality by data partitioning, data replication and heuristic or cost-based join reordering [10, 15].

MapReduce frameworks like Hadoop and Spark propose a variety of physical data layers with different data storage and access methods. The choice of a specific data layer for building a distributed RDF store is generally driven by the available data abstraction APIs and their ease of use for implementing specific data partitioning and join optimization strategies. In this article we first evaluate the influence of different distributed data layer implementations on the performance of distributed join plans. The generic Spark platform supports this evaluation within a uniform optimized software environment by eliminating low-level performance side-effects related to using different technologies. The second contribution is a new cost-based framework for the distributed processing of SPARQL graph pattern queries on top of Spark.

Our main contribution is an analytical and experimental evaluation of three data storage and access layers proposed by Spark (SQL, Resilient Distributed Data sets (RDD), Data Frame (DF)) for processing SPARQL basic graph patterns through distributed join plans. The analytical evaluation is based on a simple cost model for the two main distributed join operations (partitioned and broadcast join) implemented in Spark for non-replicated partitioned data sets. We also show that this cost-model can be used to dynamically build efficient hybrid join plans combining both join operators.

GRADES'17, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3078447.3078448

2 DISTRIBUTED SPARQL PROCESSING

2.1 Basic Graph Patterns and Join Query Plans

We focus on the evaluation of basic graph patterns (BGP) which are the building blocks of more general SPARQL queries with filters, alternatives (OPTIONAL) and set operators (UNION, MINUS). Efficiently evaluating BGP expressions is essential for all SPARQL query engines and an important challenge for SPARQL query optimizers. Fig. 1(a) shows a BGP expression Q_8 aiming to retrieve the

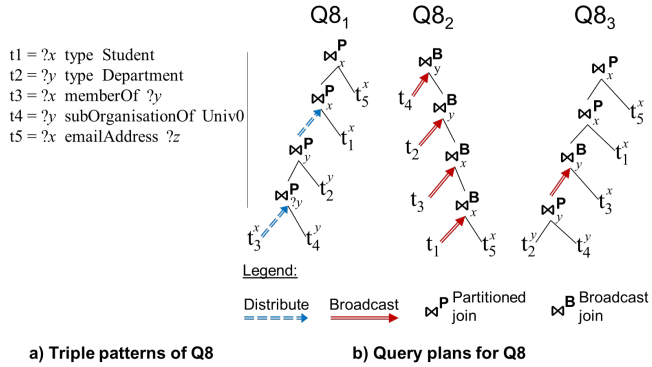


Figure 1: LUBM benchmark's $Q8$

email address $?z$ of all students $?x$ who are members of department $?y$ in university $Univ0$. Each triple pattern t_1, ..., t_5 implicitly defines a *triple selection* which computes all triples respecting this pattern. For example, t_4 filters all triples with property $subOrganisationOf$

and object $Univ0$ and *binds* variable $?y$ to the subjects of these triples. The formal query semantics of a BGP expression $e = \{t_1, ..., t_i\}$ with variables V for some RDF data set (graph) D consists in finding all variable bindings m from V to nodes and edges in D, such that $m(e)$ is a subgraph of D (subgraph isomorphism). Variables $?x$ and $?y$ are called *join variables* and define n-ary *triple pattern join*s. For example, expression $join_x(t_1, t_3, t_5)$ joins triples t_1, t_3 and t_5 on their subject, whereas $join_y(t_3, t_2, t_4)$ joins the object of t_3 with the subjects of t_2 and t_4. Both expressions can be combined to build a *join plan* $join_y(join_x(t1, t_3, t_5), t_2, t_4))$ generating all variable bindings for query Q_8. Observe that it is possible to build several equivalent plans for Q_8 t_3 and to decompose n-ary joins into several binary joins. For example, $join_x(join_x(join_y(join_y(t3, t_4), t_2), t_1), t_5)$ is another binary linear join plan for Q_8.

2.2 Data partitioning and distributed operators

Given a data set D, a query expression Q and a cluster of nodes C, the global query evaluation process is as follows: (i) the initial data set D is partitioned and distributed once over the cluster C following a predefined query-independent hash-based partitioning strategy; (ii) triple selections are evaluated locally by each node over its triple partition; (iii) joins are recursively executed following a distributed physical join plan using different physical join implementations. In the following, we provide more details on each of these steps.

Data partitioning. Due to its high efficiency, hash-based partitioning is the foundation of MapReduce-based parallel data processing infrastructures. Consider a cluster $C = (node_1, \cdots node_m)$ of m nodes and some query Q with variables V over an input data set D. Any subset V' defines a *partitioning scheme* for Q, denoted $Q^{V'}$, which describes the partitioning of the triples matched by Q with respect to the bindings of a variable subset $V' \subseteq V$. For example, $(?x\ prop\ ?y)^x$ denotes that all triples with property *prop* are partitioned by their subject, $(?x\ ?p\ ?y)^p$ denotes a vertical partitioning by property type, $(?x\ ?p\ ?y)^x$ denotes a horizontal partitioning by subject and $(?x\ ?p\ ?y)^{x\ y}$ denotes a partitioning by subject and object. Partitioning schemes are independent of the partitioning process itself and their main purpose is to decide which triple joins can be evaluated locally within the partitioned data set. In the following, we suppose that all triples of the input data set D are partitioned by their subject. The partitioning scheme defines which joins can be processed locally and which joins induce data transfer cost between nodes.

Triple selection. Given a triple pattern t, the triple selection algorithm consists in scanning the whole input data set D (no indexing assumption). All triple selections are evaluated locally on each cluster node and generate no data transfer. We rely on the semantic encoding that we proposed in [7] to perform such selections. Triple selection preserves the partitioning schemes of their input, *i.e.*, the result of a triple selection has the same partitioning as the input data set. For instance, considering query Q_8 (Fig. 1(a)) over a triple set D partitioned by subject, we obtain the following partitioning schemes for each triple selection query: t_1^x, t_2^y, t_3^x, t_4^y, t_5^x.

Partitioned Join: $Pjoin$**.** Let $Q = join_V(q_1^{p_1}, q_2^{p_2})$ be a join query, with q_i a triple pattern or a sub-query. The *partitioned join* operator, henceforth denoted $Pjoin_V(q_1^{p_1}, q_2^{p_2})$, repartitions and distributes, when necessary, the input data over the bindings of all variables in V (*i.e.*, it shuffles on V) then computes the join result for each partition in parallel as detailed in Algorithm 1 of Appendix A. We distinguish three cases depending on p_i values : (i) $p_1 = V \wedge p_2 = V$; the join is local since every q_i is already partitioned on the join key V. This case generates no data transfer. (ii) $p_1 = V \wedge p_2 \neq V$; the result of q_2 is shuffled on V before evaluating the join. (iii) $p_1 \neq V \wedge p_2 \neq V$. Every q_i's result is shuffled on V before evaluating the join.

The result of Q is partitioned on V, denoted Q^V. The corresponding transfer cost is:

$$\sum_{1 \leq i \leq 2 \wedge p_i \neq V} Tr(q_i)\ with\ Tr(q_i) = \theta_{comm} * \Gamma(q_i)$$

where $\Gamma(q)$ is the result size of a given sub-query q and θ_{comm} is the unit transfer cost.

Broadcast Join. The *broadcast join*, denoted $Brjoin_V(q_1^{p_1}, q_2^{p_2})$, consists in sending the query result of q_1 to *all* compute nodes, as detailed in Algorithm 2 of Appendix A. Without loss of generality, we assume that q_2 is the target sub-query, excluded from the broadcast step, and has a larger size than q_1. The broadcast join does not consider the partitioning of its arguments and preserves the partitioning of the target query, *i.e.*, the result of the broadcast join has the same partitioning as q^{p_2}. The corresponding transfer cost is:

$$(m-1) * Tr(q_1)$$

where m is the number of nodes and $Tr(q_i)$ is defined as before.

3 SPARQL PROCESSING ON SPARK

Apache Spark [14] is a main-memory cluster computing engine which enables parallel computations on unreliable machines and automatic locality-aware scheduling. This efficiency is mainly due to two complementary distributed main-memory data abstractions: (i) Resilient Distributed Data sets (RDD), a distributed, lineage supported fault tolerant data abstraction for in-memory computations and (ii) Data Frames (DF), a compressed and schema-enabled data abstraction. Both data abstractions ease the programming task by natively supporting a subset of relational operators like *project* and *join*. These operators enable the translation and processing of high-level query expressions (*e.g.*, SQL, SPARQL).

On top of RDD and DF, Spark proposes two higher-level data access models, GraphX and Spark SQL, for processing semi-structured data in general, and SPARQL queries over RDF data in particular. Spark GraphX[1] is a library enabling the manipulation of graphs through an extension of Spark's RDD and follows a vertex-centric computation model which is dedicated to perform highly-parallel iterative algorithms on graphs. This processing model is not adapted to set-oriented graph pattern matching and is not considered in our evaluation. Spark SQL [4] allows for querying structured data stored in DFs. Its optimizer, Catalyst [4] can be used with the DF API and a Domain Specific Language (DSL) for formulating queries.

We propose four approaches to enable SPARQL query processing on top of Spark SQL, RDD and DF. Moreover, we investigate how far each method supports the algorithms presented in Sec. 2 to evaluate joins. The first three approaches are worth mentioning because they are used in many state-of-the art distributed SPARQL processing solutions. We consider them as baselines. The last approach is part of our contributions.

[1] https://spark.apache.org/docs/latest/graphx-programming-guide.html

3.1 SPARQL SQL

The SPARQL SQL method consists in rewriting a given SPARQL query Q into a SQL query Q' which is evaluated by the Spark SQL engine [4]. The execution plan of Q' is determined by the embedded Catalyst optimizer using the Spark DF data abstraction which applies the $Brjoin$ algorithm introduced in Sec. 2.2 . It generates a join plan which broadcasts all triple patterns, except the last one which is the target pattern. In our experiments with Spark SQL version 1.5.2, we observed that, when a query contains a chain of more than two triple patterns, a cartesian product is used rather than a join. Consider 3 triples patterns $t_1 = (a, p1, x), t_2 = (x, p2, y)$ and $t_3 = (y, p3, b)$, and the query $join_y(join_x(t1, t2), t3)$. Then, for the corresponding SQL expression, Catalyst generates the physical plan $Q_1 = Brjoin_{xy}(Brjoin\_(t_1, t_3), t_2)$ which computes a cross product between t_1 and t_3 before joining with t_2. This obviously is less efficient than, for example, plan $Q_2 = Brjoin_y(Brjoin_x(t_1, t_2), t_3))$. Fig. 1b shows Q_{8_2}, a 5-way join plan which implies to broadcast the results of t_1, t_3, t_2, and t_4.

3.2 SPARQL RDD

The SPARQL RDD approach consists in using the Spark RDD data abstraction and specifically the $filter$ and $join$ methods of the RDD class for evaluating SPARQL queries over large triple sets. Every logical join translates into a call to the $join$ method. which implements the $Pjoin$ algorithm introduced in Sec. 2.2. This strategy translates each join into a $PJoin$ operator, following the order specified by the input logical query, and recursively merges successive joins on the same variable into one n-ary $Pjoin$. This ends up with a sequence of (possibly n-ary) joins on different variables. The result of the first n-ary join on variable, say v_1, is distributed before processing the next join on variable, say v_2, and so on. Fig. 1 shows the *PJoin plan* of Q_8: $Q_{8_1} = Pjoin_x(Pjoin_y(t_3^x, t_2^y, t_4^y,)^y, t_1^x, t_5^x)$. It distributes t_3 triples based on y, then joins them with t_2 and t_4 on y. The result is shuffled on x to be joined with t_1 and t_5 on x. The overall transfer cost of Q_{8_1} is $\theta_{comm} \cdot (Tr(t_3) + Tr(join_y(t_4, t_2, t_3)))$.

It evaluates star pattern (sub-) queries locally (*i.e.*, no shuffle) when the input data is partitioned by the join variable, which is obviously efficient. However, it lacks efficiency when a broadcast join is cheaper, *e.g.*, join a small with large data set. Observe that SPARQL RDD always reads the entire data set for each triple pattern evaluation. This is remedied by merging multiple triple selections (Sec. 3.4).

3.3 SPARQL DF

Spark Data Frame (DF) provides an abstraction for manipulating tabular data through specific relational operators. Translating a SPARQL query using the DF DSL is straightforward: triple selections translate into DF *where* operators whereas SPARQL n-ary join expressions are transformed into trees of binary DF *join* operators. The main benefit of using this approach comes from the columnar, compressed in-memory representation of DF. The advantages are twofold. First it allows for managing larger data sets (*i.e.*, up to 10 times larger compared with RDD) for a given memory space, and second, DF compression saves data transfer cost. DF uses a cost-based join optimization approach by preferring a single broadcast join to a sequence of partitioned join if the size of the data set is less than a given threshold. This achieves efficient query processing when joining several small data sets with a large one. However we could observe two important drawbacks in applying the SPARQL DF approach. The first drawback comes from the fact that DF only takes into account of the size of the input data set for choosing $Brjoin$. By this, DF does not efficiently handle very frequent join expressions $join(s, t)$ where s is a highly selective filtering expression over a large data set. In that case, $Brjoin$ would be more efficient since it would avoid the data transfer for pattern t (see Sec. 5 for performance comparison between partitioned and broadcast joins). Example Q_{8_2} illustrates the processing of Q_8 through the DF layer.

The second drawback is that SPARQL DF (up to version 1.5) does not consider data partitioning and there is no way to declare that an attribute among (s, p or o) is the partitioning key. Consequently, partitioned joins always distribute data and cause costly data transfers. This penalizes star pattern queries where the result of each triple pattern is already adequately distributed, since the query could have been answered without any transfer.

3.4 SPARQL Hybrid

The goal is to overcome the limitations found in the SPARQL SQL, RDD, and DF solutions in order to provide a more efficient SPARQL processing solution on top of Spark. In particular, we aim to: (i) take into account the current data partitioning to avoid useless data transfers, (ii) enable data compression provided by the DF layer to save data transfers and manage larger data sets, and (iii) reduce the data access cost of self-join operations.

As emphasized in our evaluation (see Sec.5), this SPARQL Hybrid strategy allows for combining $PJoin$ and $BrJoin$. First, this allows the query optimizer to exploit knowledge about the existing data partitioning for combining local partitioned joins with broadcast joins. For example, if a subject-based partitioning scheme has been applied to the data set, an optimal join plan for a "snowflake" query pattern like Q_8 might join the result of a set of local partitioned joins ("star" sub-queries) through a sequence of broadcast joins: Plan Q_{8_3} of Fig. 1 first joins t_4 with t_2 on y without any transfer because t_4 and t_2 are adequately partitioned on their subject y. Then, it broadcasts the result and joins it on y with t_3 preserving the partitioning of t_3 on x. Finally it locally joins the result with the remaining patterns t_1 and t_5 which are also adequately partitioned on their subject x. Plan Q_{8_3} has a lower transfer cost than the plans generated by the other planning strategies.

We rely on our cost model to demonstrate that combining the $BrJoin$ and *distributed* $PJoin$ algorithms might also yield more efficient plans than all other plans using only one distributed join algorithm. We highlight an example where a plan combining both $Pjoin$ and $BrJoin$ algorithms is beneficial. Fig. 2 shows three join plans for LUBM query Q_9:

$$Q_{9_1} = Pjoin_y(t_1^x, Pjoin_z(t_2^y, t_3^z)^z)^y \quad (1)$$

$$Q_{9_2} = Brjoin_z(t_3^z, Brjoin_y(t_2^y, t_1^x)^x)^x \quad (2)$$

$$Q_{9_3} = Pjoin_y(t_1^x, Brjoin_z(t_3^z, t_2^y)^y)^y \quad (3)$$

Plan Q_{9_1} is composed of two distributed partitioned joins, plan Q_{9_2} is composed of two broadcast joins, whereas Q_{9_3} is a hybrid plan combining a broadcast join on y and a distributed partitioned join on z. Suppose the following order on the size of the patterns

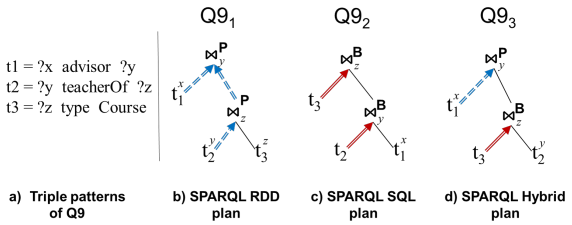


Figure 2: LUBM query Q_9

$\Gamma(t_1) > \Gamma(t_2) > \Gamma(t_3)$ and $\Gamma(join_y(t_1, t_2)) > \Gamma(join_z(t_2, t_3))$. Then it is easy to see that Q_{9_1} is the optimal partitioned join plan and Q_{9_2} the optimal broadcast join plan. The cost of these plans is:

$$cost(Q9_1) = \theta_{comm} * (\Gamma(t_1) + \Gamma(t_2) + \Gamma(join_z(t_2, t_3))) \quad (4)$$

$$cost(Q9_2) = \theta_{comm} * (m-1) * (\Gamma(t_2) + \Gamma(t_3)) \quad (5)$$

$$cost(Q9_3) = \theta_{comm} * (\Gamma(t_1) + (m-1) * \Gamma(t_3)) \quad (6)$$

Based on that cost model, the best plan depends on the number of machines. For small m, Q_{9_2} wins because it broadcasts small sized triple patterns. For large m, Q_{9_1} wins because it does not broadcast any data. In between, we infer the following two inequalities specifying the range of values for which the Q_{9_3} hybrid plan wins:

$$\Gamma(t_1) < (m-1) * \Gamma(t_2) \text{ and } (m-1) * \Gamma(t_3) < \Gamma(t_2) + \Gamma(join_z(t_2, t_3))$$

If m is "high enough" then distributing the large sized t_1 is cheaper than broadcasting the medium sized t_2. If m is not "too high" then broadcasting the small sized t_3 is cheaper than distributing both the medium sized t_2 and the result of $join(t_2, t_3)$.

We have implemented a simple dynamic greedy SPARQL optimization strategy using this cost model which introduces a fine-grained control of the query evaluation plan at the operator level. The initial input plan is a set of triple patterns with a size estimation for each pattern (necessary statistics are generated during the data loading phase). An evaluation step then consists in (1) choosing the pair of sub-queries and the join operator which generate the minimal cost using our cost-model, (2) executing the obtained join expression and (3) replacing the join arguments by the join expression and an exact result size estimation. This step is iteratively executed until there remains a single join expression in the input plan. This strategy is implemented in both SPARK data abstraction layers, RDD and DF. For the RDD abstraction (which does not support the *Brjoin* natively) we decompose the *Brjoin* operator into two jobs, one for broadcasting the data and the other for computing the join result based on the broadcast data using the *mapPartition* RDD transformation method[2]. For the DF abstraction, to ensure that the *BrJoin* operator runs consistently according to the hybrid choice, we had to switch-off the less efficient threshold-based choice condition of the Catalyst optimizer.

Merging multiple triple selections. Consider a query $Q = \{t_1, \cdots, t_n\}$ composed of n triple patterns. Since all t_i's are expressed over the same data set D, there are opportunities to save on access cost for evaluating Q. The basic idea is to replace n scans over the whole data set D by a single scan over the whole data set and k scans over a much smaller sub-set. For this, we first rewrite the selections in Q into a single selection $S = \sigma_{c_1 \vee \cdots \vee c_n}(D)$ where c_i is the select condition of t_i, which returns all triples $\bigcup_{i=1}^{n} t_i$ necessary for evaluating Q. To implement this *merged access* approach we added a preliminary step to persist the covering subsets in main-memory. Our experiments will show the benefit of this approach.

[2]http://spark.apache.org/docs/latest/programming-guide.html#transformations

3.5 Qualitative Comparison

The different methods presented before can be compared according to four dimensions:

- **Co-partitioning** : All methods except SPARQL DF (version 1.5) and SPARQL SQL exploit existing data partitioning information to join triples partitioned on the join key locally.
- **Join algorithm** : SPARQL RDD only uses partitioned join, SPARQL DF and SPARQL SQL can combine partitioned joins with a single broadcast join, whereas both hybrid implementations can combine an arbitrary number of partitioned and broadcast joins.
- **Merged access**: Both hybrid methods (Hybrid RDD and Hybrid DF) implement multiple (disjunctive) triple filters which can be processed through a single data scan.
- **Data compression**. All DF based methods (SPARQL DF, SQL and Hybrid DF) use data compression and allow for managing ten times larger data sets than RDD, at equal memory capacity.

In conclusion, the *SPARQL Hybrid* method offers equal or higher support for all the considered properties. Interestingly, *SPARQL Hybrid* fits with both Spark data abstractions, RDD and DF, because the underlying logical join optimization is separated from the physical data representation.

4 RELATED WORKS

CliqueSquare [8] aims at maximizing local joins, but replicates the whole data set 3 times which is not applicable to a main-memory approach. S2RDF [12] is built on Spark and uses its SQL interface to execute SPARQL queries. Its main goal is to address efficiently all SPARQL query shapes. Its data layout corresponds to the vertical partitioning (VP) approach, *i.e.*, triples are distributed in relations of two columns (one for the the subject and one for the object) corresponding to RDF properties. So-called ExtVP relations are computed at data load-time using semi-joins, to limit the number of comparisons when joining triple patterns. Considering query processing, each triple pattern of a query is translated into a single SQL query and the query performance is optimized using the set of statistics and additional data structures computed during this pre-processing step. The data pre-processing step generates an important data loading overhead which might be up to 2 orders of magnitude larger than the subject-based partitioning without replication of our solution. The AdPart system [2] implements a main memory SPARQL engine using MPI (Message Passing Interface) data transfer and lacks fault tolerance (contrary to Spark). This engine uses a distributed semi-join operator to limit data transfer for selective joins over large sub-queries by combining adapted partitioned and broadcast join variants. It could be interesting to study this new operator within our framework. Distributed multiway join

processing in general has been the topic of many research papers since decades [11] and we will cite only some more recent representative contributions about parallel distributed multiway joins over partitioned data. In [1], a solution is presented for the computation of multi-join queries in a single communication round. The algorithm was originally designed for the MapReduce approach, thus justifying the importance of limiting communication costs which are associated to a high I/O costs. The authors of [5] have generalized this single-communication n-ary join problem over a fixed number of servers and designed a new algorithm named HyperCube by providing lower and upper communication bounds. HyperCube is also at the origin of an implementation presented in [6]. This work is a promising approach for evaluating SPARQL queries in a MapReduce setting where the number of rounds has to be restricted. We have chosen a different setting, where data is in main memory and partitioned, thus reducing the whole data transfer cost independently of the number of rounds (join tree depth).

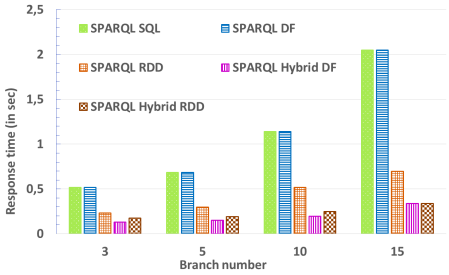
5 EXPERIMENTAL EVALUATION

We validate the query processing methods of Sec. 3 with two synthetic and three real world workloads including star, chain, and snowflake queries. Both synthetic data sets, the Lehigh University Benchmark (LUBM) [9] and the Waterloo SPARQL Diversity Test Suite (WatDiv) [3], provide a data generator and a set of queries. The real world data sets correspond to DBPedia, Wikidata and DrugBank RDF dumps. All data sets are partitioned by the triple subjects to optimize star queries. The evaluation was conducted on a 18 DELL PowerEdge R410 cluster interconnected by a 1GB/s Ethernet network. Each machine runs a Debian Linux distribution and has 2 Intel Xeon E5645 processors, each constituted of 6 cores with 2.40GHz clock-rate and hyper threading (two threads). We used Spark version 1.6.2 and implemented all experiments in Scala with a configuration of 300 cores and 50GB of RAM per machine. More experimentation details are available on the companion web site[3].

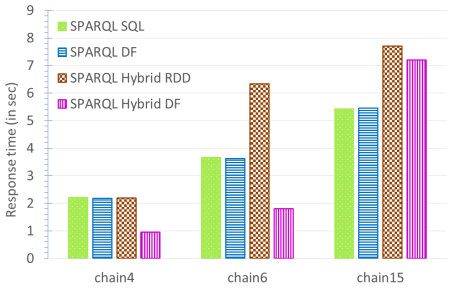
Star Queries. This experiment was conducted over the DrugBank knowledge base (505k triples) which contains high out-degree nodes describing drugs. A first practical use case is to search for a drug satisfying multi-dimensional criteria and we defined four star queries with out-degrees ranging from 3 to 15. The query response times are reported in Fig. 3(a). SPARQL SQL and DF ignore the actual data partitioning and generate unnecessary data transfers. SQL and DF are about 2.2 times slower than SPARQL RDD and Hybrid which evaluate a star query locally without any transfer costs. Moreover, SPARQL Hybrid outperforms SPARQL RDD because of the merged multiple triple selection operator scanning the dataset only once per query instead of once per star branch.

Property Chain Queries. This experiment was done over the DBPedia knowledge base (77.5M triples) and chain queries with a length ranging from 4 to 15. We report the query response times in Fig. 3(b). Chain queries *chain*4 and *chain*6 contain large (not selective) triple patterns followed by small (selective) ones. These "large.small" sub-chains should be evaluated by broadcasting the smaller pattern instead of shuffling the larger one. The strength of SPARQL Hybrid DF is here to estimate the patterns' selectivity at runtime and more accurately than SPARQL DF. This allows

[3]https://sites.google.com/site/sparqlspark/home



(a) Star queries



(b) Chain queries

Figure 3: Query response time wrt. evaluation strategies on real world data sets

SPARQL Hybrid DF to choose broadcast joins for this case, whereas SPARQL DF inaccurately estimated the pattern selectivities and favored partitioned joins which caused large transfer costs.

SPARQL Hybrid DF recursively chooses the lowest cost join based on the size estimations of the intermediate results and the remaining triple patterns. This might lead to a suboptimal plan as shown for chain query *chain*15 (SPARQL DF only uses partitioned join which is more efficient). In this specific query, the first triple pattern (say t_1) and the following one (say t_2) are large compared to the other ones, but joining t_1 with t_2 yields a very small intermediate result. However, this knowledge is not available before evaluating the join and cannot be exploited by SPARQL Hybrid DF.

Snowflake Queries. First, we focus on the most complex

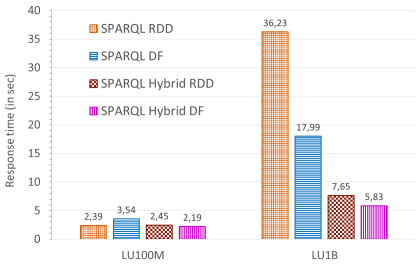


Figure 4: LUBM query Q8

snowflake query of the LUBM benchmark (Q_8) over LUBM100M (133M triples) and LUBM1B (1.33B triples). The evaluation plans

for Q_8 have been introduced in Sec. 2 and we report the response times in Fig. 4. Q_8 did not run to completion with SPARQL SQL. The evaluation plan contained a cartesian product that was prohibitively expensive. This emphasizes that the Spark's Catalyst optimizer strategy to replace two joins by one cartesian product should be applied more adequately by taking into account the actual transfer cost. SPARQL DF and SPARQL RDD confirm that working with compressed data is beneficial as soon as the data set is large enough. Although SPARQL DF ignores data partitioning, thus distributing more triples (320M instead of 104M triples for the partitioning-aware approach), its transfer time is lower than SPARQL RDD, thanks to compression.

The major experimental result is that SPARQL Hybrid outperforms existing methods by a factor of *2.3* for compressed (DF) and *6.2* for uncompressed (RDD) data. This is mostly due to reduced transfers (only few hundred triples instead of over one hundred million triples for LUBM1B). SPARQL Hybrid also saves on the number of data accesses: 2 against 3 and 5 for resp. SPARQL RDD and SPARQL DF.

Comparison with S2RDF. We finally compare our Hybrid approach with the state of the art S2RDF [12] solution which outperforms most other existing distributed SPARQL processing solutions. We conducted the S2RDF comparative experiments over the same WatDiv 1 billion triple data set on a cluster with approximately similar computing power than the one used in the S2RDF evaluation (we used 48 cores in our experiment against 50 cores used in the S2RDF experiments). Our main goal is to show that our solution is complementary and can be combined with the S2RDF approach. For this, as a baseline we first selected three representative queries from the WatDiv query set, one for each category: S_1 is a star query, F_5 a snowflake one, and C_3 a complex one. We executed S_1, F_5 and C_3 over one large data set containing all the triples (*i.e.*, without S2RDF VP fragmentation), using SPARQL SQL and SPARQL Hybrid strategies. Then, we split the data set according to the S2RDF VP approach (*i.e.*, one data set per property) and ran the queries using SPARQL SQL along with the S2RDF ordering method, and SPARQL Hybrid strategies (see Fig. 5).

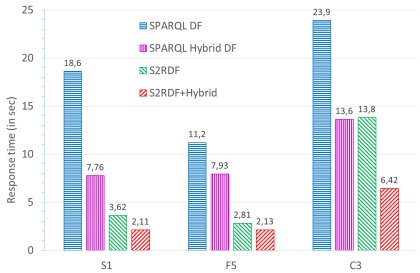


Figure 5: WatDiv queries on 1B triples

Our SPARQL Hybrid solution outperforms the baseline SPARQL SQL and the S2RDF solution by a factor of 2 which is encouraging. This performance gain mainly comes from reduced data transfer costs saving 483MB for S1, 284MB for F_5, and 1.7GB for C_3. Note that while reproducing the S2RDF experiments, we get response times more than twice faster than those reported in [12] (*e.g.*, 3.6 sec instead of 8.8 sec for query S_1) and our *1.72* minimal improvement ratio is a fair comparison. This highlights that our approach easily combines with S2RDF to provide additional benefit. We did not compare our approach with the concept of ExtVP relations of S2RDF's solution, since it comes at high pre-processing overhead (17 hours for pre-processing 1 billion triples) which does not comply with our objectives of reducing data pre-processing and loading cost.

6 CONCLUSION

In this article, we present a comprehensive study comparing four SPARQL query processing strategies for Apache Spark RDF data stores. These strategies have been implemented and evaluated over different benchmark queries and data sets. The obtained results emphasize that hybrid query plans combining partitioned and broadcast joins improve query performance in almost all cases. Our solution is orthogonal to recent state-of-the-art map-reduce based SPARQL query processing approaches. For example, it naturally fits into the recent Spark-based S2RDF system to improve its performance. As future work, we plan to implement our approach as part of a full-fledged SPARQL query engine. More theoretically, we intend to explore more deeply the interaction between data partitioning schemes and distributed join algorithms as part of a general distributed join optimization framework.

REFERENCES

[1] Foto N. Afrati and Jeffrey D. Ullman. 2010. Optimizing joins in a map-reduce environment. In *EDBT*. 99–110.

[2] Razen Al-Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. 2016. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *VLDB J.* 25, 3 (2016), 355–380.

[3] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified Stress Testing of RDF Data Management Systems. In *ISWC*. 197–212.

[4] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*. 1383–1394.

[5] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in parallel query processing. In *ACM PODS*. 212–223.

[6] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. In *SIGMOD*. 63–78.

[7] Olivier Curé, Hubert Naacke, Tendry Randriamalala, and Bernd Amann. 2015. LiteMat: a scalable, cost-efficient inference encoding scheme for large RDF graphs. In *IEEE Big Data*. 1823–1830.

[8] François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge-Arnulfo Quiané-Ruiz, and Stamatis Zampetakis. 2015. CliqueSquare: Flat plans for massively parallel RDF queries. In *IEEE ICDE*. 771–782.

[9] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.* 3, 2-3 (2005), 158–182.

[10] Zoi Kaoudi and Ioana Manolescu. 2015. RDF in the clouds: a survey. *VLDB Journal* 24, 1 (2015), 67–91.

[11] Hongjun Lu, Ming-Chien Shan, and Kian-Lee Tan. 1991. Optimization of Multi-Way Join Queries for Parallel Execution.. In *VLDB*. 549–560.

[12] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. 2016. S2RDF: RDF Querying with SPARQL on Spark. *Proc. of the VLDB Endowment* 9, 10 (2016), 804–815.

[13] Tom White. 2009. *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.

[14] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[15] M. Tamer Özsu. 2016. A survey of RDF data management systems. *Frontiers of Computer Science* (2016), 1–15.

A JOIN ALGORITHMS

A.1 Partitioned Join

Let $Q = join_V(q_1^{p_1}, \cdots, q_n^{p_n})$ be an n-ary join query. The *partitioned join* operator, denoted $Pjoin_V(q_1^{p_1}, \cdots, q_n^{p_n})$, partitions (when necessary) the input data over the bindings of all variables in V and computes the join result for each partition in parallel. Let d_i be the result of q_i, and d_{ij} be the chunk of d_i on $node_j$ ($1 \leq j \leq m$). The partitioned join algorithm evaluates the query result in four steps as detailed in Algorithm 1. After reading the input set d_i of each sub-query (lines 3-5), partition (if necessary) each d_i based on the join key V into m partitions (lines 6-7). The third step transfers (*i.e.*, shuffles) the data to the m target nodes (lines 8-9) such that each node is responsible for computing the join for some values of V (lines 10-11). The result of Q is partitioned on V, denoted Q^V.

Algorithm 1 Partitioned Join

1: **Input**: $\{q_1^{p_1}, q_2^{p_1}\}$, join variables V
2: **Output**: result fragment $Result_j$ on each node $node_j$
▷ Evaluate and shuffle sub-query results
3: **for all** q_i **do**
4: **for all** $node_j$ **do**
5: $d_{ij} \leftarrow$ evaluate q_i on node $node_j$
6: **if** $p_i \neq V$ **then**
7: repartition d_{ij} on V into $\{d_{ij1}, \cdots, d_{ijm}\}$
8: **for all** $node_k \neq node_j$ **do**
9: transfer d_{ijk} from $node_j$ to $node_k$
▷ Compute join locally on each node
10: **for all** $node_j$ **do**
11: $Result_j^V \leftarrow (\bigcup_{x=1}^{m} d_{1xj}) \bowtie (\bigcup_{x=1}^{m} d_{2xj})$

A.2 Broadcast Join

The *broadcast join*, denoted $Brjoin_V(q_1^{p_1}, \cdots, q_n^{p_n})$, consists in sending the query result d_i of *each* sub-query q_i except one, called the target query, to *all* nodes $node_j$. Without loss of generality, we assume that q_n is the target sub-query, excluded from the broadcast step, and has the largest result size among all sub-queries. The corresponding steps are detailed in Algorithm 2.

Algorithm 2 Broadcast Join

1: **Input**: $\{q_1^{p_1}, q_2^{p_2}\}$, join variables V
2: **Output**: result fragment $Result_j$ on each node $node_j$
▷ Evaluate and broadcast q_1 result
3: **for all** $node_j$ **do**
4: $d_{1j} \leftarrow$ evaluate q_1 on node $node_j$
5: **for all** $node_k \neq node_j$ **do**
6: transfer d_{1j} from $node_j$ to $node_k$
▷ Compute join locally on each node
7: **for all** $node_j$ **do**
8: $d_{2j} \leftarrow$ evaluate q_2 on node $node_j$
9: $Result_j^{p_2} \leftarrow \bigcup_{y=1}^{m} d_{1y} \bowtie d_{2j}$