SPARQL query processing with Apache Spark

Hubert Naacke\(^1\), Olivier Curé\(^2\), Bernd Amann\(^1\)

\(^1\) Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris.
\{firstName.lastname\}@lip6.fr

\(^2\) LIGM (UMR 8049), CNRS, ENPC, ESIEE, UPEM, F-77454, Marne-la-Vallée, France.
olivier.cure@u-pem.fr

Abstract. The number of linked data sources and the size of the linked open data graph keep growing every day. As a consequence, semantic RDF services are more and more confronted to various “big data” problems. Query processing is one of them and needs to be efficiently addressed with executions over scalable, highly available and fault tolerant frameworks. Data management systems requiring these properties are rarely built from scratch but are rather designed on top of an existing cluster computing engine. In this work, we consider the processing of SPARQL queries with Apache Spark. We propose and compare five different query processing approaches based on different join execution models and Spark components. A detailed experimentation, on real-world and synthetic data sets, emphasizes that two approaches tailored for the RDF data model outperform the other ones on all major query shapes, i.e., star, snowflake, chain and hybrid.

1 Introduction

The Semantic Web is increasing very rapidly and generates large volumes of RDF data \([1]\). Semantic Web data processing tasks like semantic knowledge discovery and data integration generally rely on the access to several billions of RDF triples contained in the Linked Open Data (LOD) cloud or other open data and knowledge sources like Schema.org. RDF data management is a central component of these tasks. With data sets ranging from hundreds of millions to billions of RDF triples, RDF data management is expected to meet properties such as scalability, high availability, automatic work distribution and fault tolerance. Supporting efficient SPARQL query processing in such a context becomes an important challenge and we are convinced that without robust and efficient RDF stores, it is the whole Semantic Web vision that is at stake.

The features expected from modern RDF stores are reminiscent of the Big data trend and solutions implementing such stores from scratch are rare due to the enormous amount of development effort they require. Instead, many RDF data processing systems prefer to rely on an existing cluster computing engine based on the MapReduce approach \([5]\). Nevertheless, these engines cannot be considered as full-fledged data management systems \([11]\) and even integrating an efficient query processor on top of them is challenging. In particular, data storage and communication costs generated by the evaluation of joins over distributed data need to be addressed cautiously. Furthermore, new data sets of ever increasing size need fast loading and storage strategies to
shorten data ingestion time before querying. Thus, we often have to trade data-to-query storage locality for better timeliness.

Cluster-based data and query processing is highly distributed and complex SPARQL queries over large RDF graphs have to combine a lot of distributed pieces of data through join operations. Choosing the adequate join strategy that minimizes data transfer is difficult because it often depends on a priori knowledge about the data distribution and the query structure that is difficult to obtain under big data assumptions. To compensate the absence of this knowledge one needs a cautious just-in-time investigation among various ways to evaluate joins.

Several systems addressed these problems using the MapReduce cluster computing approach, e.g., SHARD [10], nHopDB [6] and SHAPE [8]. Nonetheless, these systems did not reach expected performances due to MapReduce’s inability to tackle interactive or iterative tasks that we are aiming for in a full-fledged distributed RDF store. The main reason is its lack of an abstraction for efficiently handling the main memory distributed over the cluster’s machines. Recent systems, such as Apache Spark [16] or Apache Flink[3] have been designed to address these drawbacks.

Our contributions presented in this article are the following: (1) we propose a formalization for evaluating the cost of SPARQL query processing in a distributed setting (Section 3), (2) we design and compare five Spark-based SPARQL query processing solutions using our theoretical framework (Sections 4 and 5), (3) we finally validate our framework and evaluate these solutions by an experimental evaluation over real-world and synthetic data sets (Section 6). To the best of our knowledge, this is the first theoretical and experimental evaluation of Apache Spark for processing SPARQL queries at this level of detail.

2 Preliminaries: SPARQL, Hadoop and Apache Spark

RDF and SPARQL RDF is a schema-free data model that supports the description of data on the Web. Assuming disjoint infinite sets I (RDF IRI references), B (blank nodes) and L (literals), a triple \((s,p,o) \in (I \cup B) \times I \times (I \cup B \cup L)\) is called an RDF triple with \(s\), \(p\) and \(o\) respectively the subject, predicate and object. Assume that \(V\) is an infinite set of variables and that it is disjoint with \(I\), \(B\) and \(L\). We can recursively define a SPARQL[4] triple pattern as follows: (i) a triple \(tp \in (I \cup V) \times (I \cup V) \times (I \cup V \cup L)\) is a (simple) triple pattern, (ii) if \(tp_1\) and \(tp_2\) are triple patterns, then \((tp_1, tp_2)\) represents a group of triple patterns that must all match, \((tp_1 \text{ OPTIONAL } tp_2)\) where \(tp_2\) is a set of patterns that may extend the solution induced by \(tp_1\), and \((tp_1 \text{ UNION } tp_2)\), denoting pattern alternatives, are triple patterns and (iii) if \(tp\) is a triple pattern and \(C\) is a built-in condition then the expression \((tp \text{ FILTER } C)\) is a triple pattern that enables to restrict the solutions of a triple pattern match according to the expression \(C\). The SPARQL syntax follows the select-from-where approach of SQL queries. The SELECT clause specifies the variables appearing in the result set of the query, the FROM clause specifies the input data set(s) and the WHERE clause defines the triple pattern and optional sub-queries.

---

3 https://flink.apache.org/index.html
4 http://www.w3.org/TR/rdf-sparql-query/
In this article we are mainly interested in the evaluation of groups of simple triple pattern expressions without filters, alternatives and union. Efficiently evaluating such patterns is essential for all SPARQL query engines and one of the most important challenges in SPARQL query optimization. An example of such a query is shown in Figure 1. This pattern looks for the email \( z \) 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 \( S(t_i) \) which computes all triples respecting this pattern. For example \( S(t_4) \) filters all triples with property \( \text{subOrganisationOf} \) and object \( Univ0 \) and binds variable \( y \) to the subjects of these triples. Variables \( x \) and \( y \) are called join variables, since they define two triple pattern joins: \( \text{join}_x(t_1, t_3, t_5) \) and \( \text{join}_y(t_3, t_2, t_4) \) where the first join expression joins all triples on their subject, whereas the second expression, joins the object of \( t_3 \) with the subjects of \( t_2 \) and \( t_4 \). The final result is the set of bindings for \( x, y \) and \( z \) obtained after the evaluation of all selection and join operations. Observe also that at the logical level, we do not impose any ordering on the selection and join operations.

**SPARQL query processing with Hadoop** We are particularly interested in cluster-based solutions for the distributed evaluation of SPARQL expressions and in the following we will present three representative approaches using the Apache Hadoop framework. A detailed presentation and comparison of other existing approaches and RDF stores equipped with a distributed SPARQL query processor can be found in [7].

SHARD [10] is the first effort to store RDF data on top of the Apache Hadoop distributed data storage (HDFS) and processing (MapReduce) software stack. The set of triples are stored in HDFS as flat files where each line represents all the triples associated with a given subject. The SHARD query processing engine iterates over the set of triples for each triple pattern in the SPARQL query, and incrementally attempts to bind query variables to literals in the triple data, while satisfying all of the query constraints. The SPARQL query clauses are processed in several MapReduce operations, forcing high latency due to a large number of I/O operations over HDFS.

Huang et.al. [6] describes an architecture which we will denote by nHopDB and which is composed of a graph partitioner and a set of workers implemented by RDF-
3X [9] database instances. RDF data is partitioned using a standard but costly graph partitioning algorithm (METIS) extended by a simple replication strategy for reducing costly data exchange between nodes. This allows certain queries to be executed locally on a single node and completely take benefit of the high-performance RDF-3X query engine. The Hadoop engine is used to supervise query processing of queries where the answer set spans multiple partitions. In this case, nHopDB suffers from Hadoop’s start-up overhead and inherent I/O latencies, due to costly disk access operations.

The SHAPE system [8] is motivated by the limited scalability of graph partitioning-based approaches and applies simple hash partitioning for distributing RDF triples. Like in nHopDB, SHAPE replicates according to the n-hop guarantee. Hence, it takes the risk of costly inter-partition communication for query chains which are longer than their n-hop guarantee.

Apache Spark  Apache Spark [16] is a cluster computing engine that is currently considered as the Big data Apache project with the most committers. Spark can be apprehended as an extension of the MapReduce model, with Apache Hadoop being its most popular implementation. Spark enables parallel computations on unreliable machines and automatically handles locality-aware scheduling, fault tolerance and load balancing tasks. While both Spark and Hadoop are based on a data flow computation model, Spark is more efficient than Hadoop for applications requiring to reuse working data sets across multiple parallel operations. This efficiency is due to Spark’s Resilient Distributed Data set (RDD) [15], a distributed, lineage supported fault tolerant memory abstraction that enables one to perform in-memory computations (when Hadoop is mainly disk-based), or to Data Frames (DF), a compressed and schema-enabled data abstraction.

Spark’s RDD and DF APIs ease the programming task by natively supporting a large subset of relational operators like project, join and filter which are not natively supported in Hadoop. These operators enable the translation and processing of high-level query expressions (SQL, SPARQL) using any programming language supported by Spark, e.g., Java, Scala or Python as a host language.

Spark proposes two high-level data abstractions which might be used for processing semi-structured data in general, and RDF graphs in particular. Spark GraphX [13] is a library enabling the manipulation of graphs through an extension of Spark’s RDD. Being the most prone to represent graphical information, GraphX may seem to be a good candidate to perform graph operations such as graph pattern matching. In fact, GraphX is dedicated to perform parallel iterative operations on all nodes of a graph, e.g., the API proposes operations such as the Pagerank and Pregel algorithms, connected components and triangle counting. Hence this component is not adapted to navigate along a selection of graph nodes and is not considered in our proposed solutions. Spark SQL [2] is an obvious candidate for querying structured data stored in RDDs or DFs. Its optimizer, namely Catalyst [2], is claimed to improve the execution of queries. This optimizer can also be used with the DF API which proposes a Domain Specific Language (DSL) to express queries.
3 Distributed main-memory SPARQL processing

In this section, we introduce a simple physical algebra for the evaluation of SPARQL query expressions defined as follows: (i) any triple pattern \( t \) is a query expression called a selection, (ii) let \( q_1, \ldots, q_n \) be query expressions and \( V \) a set of (join) variables shared by all \( q_i \), then \( \text{join}_V(q_1, \ldots, q_n) \) is a query expression called \( n \)-ary join.

In the following, we present a simple cost model and two commonly used methods, partitioned join and broadcast join, for distributed join processing [3]. The parameters for estimating the execution and data transfer cost of each operator include the estimated number of triples \( \Gamma(x) \) of a given data set \( x \), the triple unit data access cost \( \theta_{\text{acc}} \) and the triple unit transfer cost \( \theta_{\text{comm}} \). The unit access and transfer costs depend on the used storage technology \( \text{stor} \) and the communication technology \( \text{com} \). We assume that the data set \( D \) is initially distributed across a cluster of \( M \) nodes.

**Triple Selection.** Given a triple pattern \( t \), we assume that the cost to evaluate \( t \) only depends on the size of the input data set \( D \) and the unit data access cost \( \theta_{\text{acc}} \). In particular, we assume that triple selections are evaluated locally and generate no data transfer cost: 
\[
\text{Cost}(t) = \theta_{\text{acc}} \times \Gamma(D)
\]
For the sake of simplicity, we assume also that all triple patterns are selective and the result size of a triple selection is rather small with respect to the size of the input data set.

**Partitioned join, \( \text{join}_{\text{part}} \).** The partitioned join algorithm evaluates \( \text{join}_V(q_1, \ldots, q_n) \) in two steps. It first partitions the result of each sub-query \( q_i \) on \( x \) into a set of partitions \( P(q_i) = \{ q_{i1}, \ldots, q_{id} \} \) (\( d \) is bound by the size of the cartesian product of the active variable domains \( \text{dom}(x), x \in V \)) and distributes the partitions over the cluster nodes using some hash function. Then, for each set of co-partitioned partitions \( P(Q, k) = \{ q_{ik} | i \in [1, n], k \in [1, d] \} \), it evaluates a local join \( J_k = \text{join}_{\text{local}}(q_{1k}, \ldots, q_{nk}) \). The result of \( Q \) is defined by the result of the local joins \( J_1, \ldots, J_d \) stored on the \( M \) cluster nodes.

For estimating the cost of partitioned joins, we will denote by \( q^w \) the fact that the result of \( q \) has to be partitioned and distributed over the cluster nodes (\( w \) stands for wide). In case the result of \( q \) is already distributed according to the partitioning on the join variables \( V \), \( q \) will be labeled by the symbol \( n \) (standing for narrow). The corresponding evaluation cost can be estimated as follows:
\[
\text{Cost}(\text{join}_{\text{part}}^V(q_{1}^{\tau_1}, \ldots, q_{n}^{\tau_n})) = \sum_{1 \leq i \leq n} \text{Cost}(q_i) + \sum_{1 \leq i \leq n \land \tau_i = w} \text{Tr}(q_i)
\]
where \( \text{Cost}(q_i) \) is the cost of evaluating sub-query \( q_i \), and \( \text{Tr}(q_i) \) is the cost for transferring the result of \( q_i : \text{Tr}(q_i) = \theta_{\text{comm}} \times \Gamma(q_i) \). Observe that if the results of all sub-queries \( q_i \) are already partitioned on the join variables, the join can be evaluated completely locally and there is no transfer cost.
Example 1. Consider that $D$ is partitioned on the triple subjects. The triple patterns that join on their subject do not imply any data transfer and are labeled as “narrow”. This is the case for $t_{n1}$, $t_{n3}$, $t_{n5}$ on $x$, and for $t_{n2}$, $t_{n4}$ on $y$. An evaluation plan for $Q_8$ is to distribute $t_3$ triples and join them with $t_{n2}$, $t_{n4}$ on $y$. The result is distributed to be joined with $t_{n1}$ and $t_{n5}$ on $x$. The execution plan, denoted $Q_{81}$ and shown on Figure 2, can be represented as:

$$Q_{81} = \text{join}_{\text{part}}x(\text{join}_{\text{part}}y(t_3, t_{n2}, t_{n4}))w, t_{n1}, t_{n5}).$$

If $D$ has no specific partitioning, we get the following $Q_{82}$ plan (see Figure 2):

$$\text{join}_{\text{part}}x(\text{join}_{\text{part}}y(t_3, t_{w2}, t_{w4}))w, t_{w1}, t_{w5}).$$

Broadcast join, $\text{join}^{\text{broad}}$. We consider a select expression $s$ where the $q_i$ expressions are sharing some variables $V$ with $s$. Moreover, we denote the n-ary join expression $Q$ as follows: $\text{join}_V(q_1, \cdots, q_{n-1}, s)$.

The broadcast join algorithm proceeds in two steps but does not take into account any partitioning strategy. First, it replicates the result of all sub-queries $q_i$ except one, called the target subquery, over the $M$ cluster nodes. Without loss of generality, we assume that $q_n$ is the excluded target subquery and has the highest selectivity among all sub-queries. Pattern selectivity and other statistics are collected during the data loading phase [4]. Let $q_j$ denote the result set of evaluating sub-query $q_i$ on node $j$. Then, on each node $j$, evaluate the local join $J_j = \text{join}^\text{local}_V(q_1j, \cdots, q_{n-1}, q_nj)$. The result of $Q$ is defined by the results of the local joins $J_1, \cdots, J_M$ on the $M$ cluster nodes.

The action of broadcasting the result of a sub-query is denoted with $b$ (standing for broadcast). To express that $q_n$ does not require any transfer, we use exponent $l$ (standing for local). The corresponding query cost is:

\[5\] This assumption is true for most patterns except certain patterns using property $\text{rdf:type}$, like for example $(?x \text{ rdf:type} \text{ rdf:Resource})$
Cost(\(\text{join}^\text{broad}_V(q_1^b, \cdots, q_{n-1}^b, q_n^l)\)) = Cost(q_n) + \sum_{i=1}^{n-1} Cost(q_i) + M \times Tr(q_i) \quad (1)

where Cost(q_i) and Tr(q_i) are defined as before.

Example 2. A straightforward solution is to broadcast the result of all triples patterns, except the largest one which is \(t_5\). We have (see Figure 3): \(Q_{83} = \text{join}^\text{broad}_{yx}(t_1^b, t_2^b, t_3^b, t_4^b, t_5^l)\). However this 5-way join plan still implies a huge amount of transfer because the results of \(t_3\) and \(t_1\) are large. The broadcast join is particularly efficient when joining a small triple set with a large one. To achieve this, we consider successive binary joins. We have \(Q_{84}\) as follows: \(j_1 = \text{join}^\text{broad}_y(t_4^b, t_5^l)\); \(j_2 = \text{join}^\text{broad}_y(j_1^b, t_3^b)\); \(j_3 = \text{join}^\text{broad}_x(j_2^b, t_1^l)\) and \(Q_{84} = \text{join}^\text{broad}_x(j_3^b, t_5^l)\).

4 SPARQL query processing with Apache Spark

Adopting the notations presented in Section 3 we now propose three SPARQL query processing approaches using three main data storage and access methods proposed by Apache Spark.

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 [2]. The precise execution plan of \(Q'\) is determined by the embedded Catalyst optimizer. We observed, for Spark versions up to 1.5.2, that when a query contains a chain of at least 3
triple patterns (say $t_1 \cdot t_2 \cdot t_3$), the optimizer generates the plan $P = \text{join}(t_2, \text{cross}(t_1, t_3))$ which computes a cross product between $t_1$ and $t_3$ before joining with $t_2$. This is due to the optimizer attempting to minimize the number of joins (i.e., one join for $P$ instead of two for $P' = \text{join}(\text{join}(t_1, t_2), t_3)$). Plan $P$ requires to transfer the result of a cartesian product whose size is very large compared to the result size of a join expression.

$$\text{Cost}(P) = \text{Cost}(t_1) + \text{Cost}(t_3) + \text{Cost}(t_2) + M \cdot \Gamma(t_1) + \Gamma(t_1) \cdot \Gamma(t_3) + \Gamma(t_2)$$

In most cases, the cartesian product should be avoided because $\text{Cost}(P) \gg \text{Cost}(P')$. Spark SQL may generate a plan with a prohibitive cartesian product. It minimizes the number of joins instead of minimizing the amount of transferred data. We expect future versions of Spark SQL to address this issue.

**SPARQL RDD.** The SPARQL RDD approach consists in using the Spark RDD data abstraction and specific filter and join operators for evaluating SPARQL queries over large RDF graphs. Spark RDD supports only the partitioned join algorithm $\text{join}^{\text{part}}$ and takes into account existing data partitioning to perform “narrow” joins whenever possible. For $P = \text{join}(t_1, t_2)$, we can easily show the following theorem (in general, we assume $\Gamma(t_1) \leq \Gamma(t_2)$, i.e. $\Gamma(t_2)/\Gamma(t_1) \geq 1$):

$$\text{Cost}(\text{join}^\text{part}(t_1^w, t_2^w)) < \text{Cost}(\text{join}^\text{broad}(t_1^w, t_2^l)) \iff \frac{\Gamma(t_2)}{\Gamma(t_1)} < M - 1$$

If the right-hand condition above holds for every join in a given query execution plan, SPARQL RDD is optimal in terms of data transfers. Otherwise, methods considering also broadcast joins $\text{join}^\text{broad}$ would perform better. Observe also that SPARQL RDD always reads the entire data set to evaluate a given triple pattern. We address this drawback with a novel approach detailed in Section 5.

**SPARQL DF.** Spark Data Frame (DF) provides an abstraction for manipulating tabular data through specific relational operators. Translating SPARQL query pattern expressions (Section 3) using the DF DSL is straightforward: SPARQL filtering expressions 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 data frames (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. Spark DF uses a cost-based join optimization model by choosing $\text{join}^\text{broad}$ (rather than a $\text{join}^\text{part}$) if the size of the data set is less than a given threshold. This achieves efficient query processing when joining small data sets with large ones. In our case, having only one large data set, we observed that DF never selects $\text{join}^\text{broad}$.

However we could observe two important drawbacks in applying the SPARQL DF approach. The first drawback is that Spark DF does not efficiently handle very frequent join expressions $\text{join}(s, t)$ where $s$ is a highly selective filtering expression over a large
data set. In that case, \( \text{join}^{\text{broad}} \) would be more efficient since it avoids the data transfer for pattern \( t \) (as mentioned, SPARQL DF only takes into account of the size of the underlying data set for choose \( \text{join}^{\text{broad}} \)). The second drawback is that this strategy does not consider data partitioning and there is no way to declare that an attribute among \((s, p \text{ or } o)\) is the partitioning key. Consequently, partitioned joins are always “wide” and cause costly data transfers. This penalizes the evaluation cost of star pattern queries where the result of each triple pattern is distributed, although the query could have been answered without any transfer.

**Summary** Assuming that the order of the join operations is fixed by a query plan (for instance using [12]), the main limitations of our three SPARQL query processing methods on Spark are: First, they restrict the usage of optimal join algorithms depending on the underlying structure of the data set. SPARQL RDD fails to find an efficient solution when a broadcast join is proved to be optimal (join a small with large data set), whereas SPARQL DF, it does not leverage on existing data partitioning to process join locally and triggers unnecessary transfers. Second, they do not consider plans that combine several types of joins. This leads to inefficient plans for queries composed of at least two joins where \( \text{join}^{\text{narrow}} \) performs best for the first one, and \( \text{join}^{\text{broad}} \) performs best for the second one. This concerns in particular chain pattern queries like \((t_1 \cdot t_2 \cdot t_3)\) where \( t_1 = (s, p_1, ?x) \), \( t_2 = (?x, p_2, ?y) \), and \( t_3 = (?y, p_3, o) \), and where \( t_1, t_2, t_3 \) are large and \((t_2 \cdot t_3)\) is small due to join selectivity.

5 MinScan: minimal self joins

In this section, we first present a novel join algorithm reducing the data access cost and describe its implementation on Spark.

**Minimal self-join**, denoted \( \text{join}^{\text{min}} \). The main idea of this approach is to minimize the access cost by scanning in a first pass the minimal data relevant to several sub-expressions of an n-ary join. Our \( \text{join}^{\text{min}} \) algorithm suits to the evaluation of any narrow join on a set of variables where \( D \) is partitioned on the respective join attributes of each sub-expression. An example are star pattern queries partitioned on the subject attribute.

Let \( q_1, \cdots, q_k \) be query expressions sharing a set of variables \( V \), and a “narrow” join query \( Q = \text{join}_V(q_1^n, \cdots, q_k^n) \) where every \( q_i \) is partitioned on \( V \) and can be evaluated locally. Since all \( q_i \) 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 \( k \) scans over the whole data set (one for each sub-query) by a single scan over the whole data set and \( k \) scans over much smaller sub-sets. For this, we first rewrite the selections in \( Q \) into a single selection \( U \): let \( c_i \) be the select condition of \( q_i \), and \( c = c_1 \lor \cdots \lor c_k \). We can then evaluate \( U = \sigma_c(D) \) (\( \sigma \) denotes the selection operation) which contains all triples \( \bigcup_{i=1}^k q_i \) necessary for evaluating the n-ary join. We then partition \( U \) by \( V \) (group by), and for each group \( U_V \) evaluate \( Q_V = \text{join}_V(\sigma_{c_1}(U_V), \cdots, \sigma_{c_k}(U_V)) \). The result of each \( Q_V \) contains all triples matching pattern \((q_1, \cdots, q_k)\) for each partition. The
corresponding cost of a minimal self join is dominated by the access cost for evaluating \( U \):
\[
\text{Cost}(\text{join}_{V}^{\text{min}}(q_1^n, \ldots, q_k^n)) \simeq \text{Cost}(U)
\]  

(2)

Our experiments confirm that \( \text{Cost}(U) \) essentially corresponds to the cost of scanning the whole data set (and is equivalent to \( \text{Cost}(q_i) \) for any \( q_i \)).

The \( \text{join}_{V}^{\text{min}} \) algorithm can be generalized for joining “narrow” and “broadcast” patterns together in a single operation and therefore supports snowflake queries (i.e., a star pattern extended with satellite triples) in a straightforward way. Let \( \text{join}_{V}(q_1, \ldots, q_m) \) be a join expression over a set of variables \( V \). Suppose that the data set is initially partitioned only on subsets of \( V_i \subseteq V \) shared by subsets of query expressions \( N_i \) (which can be evaluated locally on \( V_i \)). One can then iteratively evaluate a minimal local join on a first sub-expression \( N_a \) over the shared variables \( V_a \) and broadcast the result before evaluating a local join on a second disjoint subset \( N_b \) on the variables \( V_a \cup V_b \) etc. The detailed description of the \( \text{join}_{V}^{\text{min}} \) algorithm is omitted for lack of space.

**Example 3.** Suppose that all triples are partitioned on their subject. To evaluate \( Q_8 \) using minimal self join, first join \( N_A = \{t_2, t_4\} \) on \( y \), then broadcast the result and evaluate a minimal join with the remaining patterns \( N_B = \{t_3, t_1, t_5\} \) on \( x \) and \( y \). We obtain the following expression (see Figure 4):

\[
Q_8 = \text{join}_{x,y}^{\text{min}}(\text{join}_y^{\text{min}}(t_2^n, t_4^n), t_3^n, t_1^n, t_5^n)
\]

Fig. 4. Evaluation plan with MinScan join

The general cost function is defined as follows. Let \( N_1, N_2, \ldots N_k \) be sets of "narrow" sub-expressions of some query \( Q \) partitioned on disjoints subsets of variables \( V_i \) (\( 1 \leq i \leq k \)). Let \( B \) be the set of remaining "broad" sub-expressions which can not be evaluated locally. Let \( \text{Cost}(U_i) \) denote to the (constant) cost of evaluating a minimal self-join on \( N_i \) and \( Tr(N_i) \) denote the cost of broadcasting the result of the join \( \text{join}_{V_1 \cup \cdots \cup V_k}(N_1, \ldots N_k) \). Then we can define the following cost function:

\[
\text{Cost}(\text{join}_V^{\text{min}}(N_1, \ldots, N_k, B)) = \sum_{i=1}^{k} \text{Cost}(U_i) + \sum_{i=1}^{k-1} Tr(N_i) + \sum_{q_i \in B} (\text{Cost}(q_i) + Tr(q_i))
\]

The above definition makes obvious that the total data transfer cost of a complex distributed join depends on the choice and the ordering of the narrow joins. This is a standard query optimization problem which is outside the scope of this paper.
MinScan RDD and MinScan DF. We propose the following MinScan method to build on the $\text{join}^{\text{min}}$ approach. 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, (iii) choose between several join algorithms the one that minimizes transfers and (iv) reduce the data access cost of self-join operations.

We translate a SPARQL query into the RDD or DF API and rely on Spark’s extensible operators (e.g., groupByKey, flatMap, mapPartitions) to instantiate the query evaluation plan.

The cost of $\text{join}^{\text{min}}$ is, by definition, less than the cost of any other join algorithms. Thus, we translate every star pattern into our efficient n-ary $\text{join}^{\text{min}}$ algorithm. By combining the select expression of all the star’s branches into a single data access, we reduce the data access cost. We apply that principle to the other query shapes like snowflake or chain queries: first we select a subset $S$ of the data set $D$ such that $S$ is the union of the result of all the select expressions; second, we evaluate the query on $S$.

The MinScan method comprises a join optimization phase. We choose the join algorithm (either $\text{join}^{\text{broad}}$ or $\text{join}^{\text{min}}$) that minimizes the data transfer cost. To this end, we rely on the cost model defined in Section 3 where the cost of a join depends on the size of the intermediate results. To accurately estimate the size of intermediate results, without any prior knowledge on the triples pattern selectivity, we collect the size information at run-time as suggested in [14]. A typical evaluation process consists of (i) evaluating all star patterns contained in the query and (ii) finding the optimal algorithm to join the stars based on their query cost estimation.

Summary. We have proposed a high level pattern matching method which permits to execute query portions using single scans over the whole data set, and allows for transferring only the necessary compressed data across the machines. To better highlight the advantages of our methods with respect to existing ones, we give on Table 1, a synthetic view of the query processing properties of all five methods presented in this section as well as Section 4. The different dimensions are:

- **Co-partitioning.** Whether triples that are partitioned on the join key can be joined without any transfer.
- **Broadcast.** The ability to evaluate a join without moving large data sets. We use the $b, l, n$ notation of Section 3 to qualify at which level that property is supported. The pattern $(b+, l)$ (many broadcast, one local) means that the support is restricted to n-ary joins including only one large data set. The pattern $(b+, n+)$ (many broadcast, many narrow) means full support of n-ary joins due to a combined use of co-partitioning and broadcast.
- **One pass scan.** Reduced data access. Ability to process several select expressions faster than processing each one independently.
- **Hybrid join plan.** Ability to choose among several join algorithms. We qualify of “poor” the choice done by SparkSQL because it is a no-choice in most cases. Indeed, as soon as the data set is large, even if the triples pattern are highly selective, it ignores broadcast join.
- **Compression.** Whether data is compressed or not. DF allows for managing ten times larger data sets than RDD, at equal memory capacity.

Clearly, our MinScan methods offer equal or higher support for all the considered properties. Interestingly, MinScan suits to both data structures, RDD and DF, because we strive to design MinScan in an operator-oriented way. MinScan is a kind of pipeline of operators manipulating triples independently from the underlying data representation. We are confident that MinScan could easily be extended to support forthcoming Spark data structures (such as Data set or GraphFrame).

| Method          | Co-partitioning | Broadcast | One pass scan | Hybrid join | Compr. |
|-----------------|-----------------|-----------|---------------|-------------|--------|
| SPARQL RDD      | ✓               | ×         | ✓             | ×           | ×      |
| SPARQL DF       | ×               | (b+, l)   | ×             | poor        | ✓      |
| SPARQL SQL      | ×               | (b+, l)   | ×             | cross-product | ✓      |
| MinScan RDD     | ✓               | (b+, n+)  | ✓             | cost-based  | ×      |
| MinScan DF      | ✓               | (b+, n+)  | ✓             | cost-based  | ✓      |

Table 1. Qualitative analysis of 5 query processing methods

6 Experimental evaluation

**Experimental Setting** The evaluation was conducted on a cluster consisting of 16 DELL PowerEdge R410 running a Debian GNU/Linux distribution with a 3.16.0-4- amd64 kernel version. Each machine has 64GB of DDR3 RAM, a 900GB 7200rpm SATA disk and two Intel Xeon E5645 processors. Each processor is constituted of 6 cores running at 2.40GHz and allowing to run two threads in parallel (hyper threading). The machines are connected via a 1GB/s Ethernet network adapter. We used Spark version 1.5.2 and implemented all experiments in Scala version 2.10.4. The Spark configuration of our evaluation runs our prototype on a subset of the cluster corresponding to 300 cores and 50GB of RAM per machine.

**Data sets and Queries** We have selected two synthetic and two real world knowledge bases. The synthetic data sets correspond to instances of the LeHigh University Benchmark (LUBM), a well-established Semantic Web set of tools composed of an ontology, a set of queries and a data generator. The knowledge bases respectively store over 100 and 1000 million triples, resp. denoted LU100M and LU1B. The real world data sets correspond to open source DBPedia, Wikidata and DrugBank RDF dumps. The main characteristics of the ABoxes are reported in the first column of Table 2. We will validate the query processing methods presented in Section 2 and Section 5 over three common SPARQL query shapes, i.e., star, chain, and snowflake. We consider that these three forms can be easily used to handle other query shapes, e.g., triangles, trees.

More details on the evaluation can be obtained on this paper’s companion web site[6]

[6] https://sites.google.com/site/sparqlspark/home
Data compression is appealing to manage larger data sets for a given amount of memory. We quantify the space benefit of compression according to the penalty for storing and accessing compressed data. Table 2 emphasizes the size of the RDD and DF representations as well as the overheads to store and access the triples. We report the absolute increase in time due to data compression (DF creation time - RDD creation time) and access/decompression (DF scan time - RDD scan time). The table highlights that DFs occupies a much smaller space than RDD with a compression rate between 8 and 17%. The high compression rate comes at a low pre-processing cost. In fact, for the largest data set, creating a compressed DF from an existing uncompressed RDD only adds an overhead of around 3850 milliseconds (i.e., 27ms/GB). The absolute access time overhead remains below 1 sec (i.e., 804ms max) and has a minor impact when querying large data sets compared to the data transfer costs. For our largest data set containing 1.3 billion triples, it took on average 950ms to read the entire RDD data set vs. 1700 ms for the corresponding DF data set.

| Data set | # triples ×10^6 | RDD size (GB) | DF size (GB) | Size ratio in (%) | Compress. time overhead (ms) | Access time overhead (ms) |
|----------|----------------|---------------|--------------|-------------------|-----------------------------|--------------------------|
| Drugbank | 505.0          | 0.275         | 0.022        | 8.0%              | 500                         | 202                      |
| DBpedia  | 77.5           | 8.1           | 1.4          | 17.2%             | 290                         | 213                      |
| LU100M   | 133.5          | 13.9          | 1.58         | 11.4%             | 250                         | 320                      |
| Wikidata | 233.1          | 24.3          | 2.6          | 10.7%             | 1550                        | 293                      |
| LU1B     | 1334.7         | 139.2         | 18.7         | 13.4%             | 3850                        | 804                      |

Table 2. Data sets and compression rates

Query processing performance We compare the performance of the five query processing methods over star, chain, and snowflake queries. We consider data partitioning as a separate issue and adopt a simple partitioning strategy where all data sets are partitioned on their triple subject nodes.

Star queries. This experiment is conducted over the DrugBank knowledge base which contains many the high out-degree of 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 a number of branches ranging from 3 to 15. We process each query using our five SPARQL query processing approaches and report query response times in Figure 5. SPARQL SQL decides to reorder the joins only if it reduces the number of join operations which is obviously not possible for a star query containing only one join variable. Thus, SPARQL SQL generates the same evaluation plan (and cost) as SPARQL DF. Both methods ignore the actual data partitioning and broadcast the result of every triple pattern across the machines. On the opposite, SPARQL RDD, MinScan RDD, and MinScan DF are aware that the data are partitioned on the subject (i.e., the join variable) and thus decide to process the query without any data transfer. Observe that
the total costs is dominated by the transfer cost which explains why SPARQL DF is at least 2.2 times slower than the transfer-free methods. When comparing the transfer-free methods we observe that both MinScan methods are 1.4 to 2 times faster than SPARQL RDD. In fact, MinScan reads the data set only once, whereas the data access cost of SPARQL RDD is proportional the number of branches (triple selection patterns). Finally, MinScan DF slightly outperforms MinScan RDD for star queries with up to 10 branches. This is due to the way MinScan implements partitioning of intermediate results: MinScan RDD relies on a built-in `groupBy` operator whose implementation is slightly more efficient than the user defined `groupBy` operator of MinScan DF.

**Property chain queries.** This experiment is done over the DBPedia knowledge base and a set of queries with a path length ranging from 4 to 15. We report the query response times in Table 3. We first consider property patterns alternating frequent and rare properties, i.e., composed of a triple pattern with a frequent property followed by one with an infrequent property, and so on. Queries `chain4` and `chain6` belong to this category. We observe that for these queries, MinScan DF outperforms the other approaches. The relatively poor performance of SPARQL DF is due to its inability to estimate the intermediary result set’s size. In fact, SPARQL DF is not informed that some triple patterns are very selective (the result size of triple patterns ranges from one hundred to one million triples), and that it would be worth using a broadcast join rather than a partitioned join.

![Fig. 5. Star queries branching impact on processing performance](image)

| Chain query | SPARQL SQL | SPARQL DF | SPARQL RDD | MinScan DF |
|-------------|------------|-----------|------------|------------|
| chain4      | 2.21       | 2.18      | 2.20       | 0.95       |
| chain6      | 3.66       | 3.62      | 6.34       | 1.81       |
| chain15     | 5.43       | 5.45      | 7.71       | 7.20       |

*Table 3. Performance for property chain queries (in seconds)*
Our MinScan method estimates the size of intermediary results at run time. Based on that, MinScan chooses to broadcast the results of all the triples patterns except the largest one (target pattern), then evaluates the entire chain query without additional data transfers. Furthermore, as explained in Section 5, MinScan is able to get all intermediary results by accessing the data set only once (c.f. the one pass scan property). The experiments show that the overall performance gain (of the informed choice combined with the one pass scan) is about 50% compared with the other methods.

Finally, we observed that the benefit of choosing a broadcast join is increasing when the size ratio between two successive triples in the chain gets larger. As observed for query chain15, when the selectivity of every triple of the chain is high, (i.e., low volume of intermediate results) it is more efficient to run a partitioned join that broadcasts the first triple and each consecutive intermediate result. This is the main reason for SPARQL DF to outperform MinScan DF.

Snowflake queries. In this experiment, we focus on the most complex snowflake query of the LUBM benchmark (Q8). The evaluation plans for Q8 have been introduced in Sections 3 and 5 and we report the response times in Table 4.

| Data set | SPARQL SQL | SPARQL DF | SPARQL RDD | MinScan RDD | MinScan DF |
|----------|------------|-----------|------------|-------------|------------|
| LU100M   | -          | 3.54      | 2.39       | 2.19        | 2.45       |
| LU1B     | -          | 17.99     | 36.23      | 5.83        | 7.65       |

Table 4. LUBM Q8 query performance (in seconds)

Q8 does not run to completion with SPARQL SQL. The evaluation plan contains a cartesian product that is prohibitively long. This shows that the optimizer’s rule, i.e., 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 distributes more triples (320M instead of 104M triples for the partitioning-aware approach), its transfer time is less than for SPARQL RDD, thanks to compression.

The major experimental result is that MinScan outperforms existing methods by a factor of 2.3 for compressed (DF) data and 6.2 for uncompressed (RDD) data. This is mostly due to reduced transfers (only few hundred triples instead of over hundred million triples for the best existing approach). MinScan also saves data accesses since it accesses the data set only twice instead of three times for SPARQL RDD and three times for SPARQL DF.

7 Conclusion

In this paper, we present the first exhaustive study comparing SPARQL query processing strategies over an in-memory based cluster computing engine (Apache Spark). Five SPARQL execution strategies have been implemented and evaluated over different
query shapes and data sets. The results emphasize that our two new solutions, minScan RDD and DF, are the most efficient in almost all settings. Although the former is slightly more efficient than the latter due to the absence of a decompression step, it becomes interesting to switch to the DF representation when the size of RDDs almost saturates the main-memory of the cluster. Then, one can store almost 10 times more data on the same cluster size with only a small loss in performance, thanks to the minScan join strategy. In future work, we are planning to integrate further query processing optimizations.

References

1. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/Overview.html, 2014.
2. M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pages 1383–1394, ACM, 2015.
3. S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In ACM Int’l Conf. on Management of Data SIGMOD, pages 975–986, 2010.
4. O. Curé, H. Naacke, T. Randriamalala, and B. Amann. Litemat: A scalable, cost-efficient inference encoding scheme for large RDF graphs. In 2015 IEEE International Conference on Big Data, Big Data 2015, pages 1823–1830, 2015.
5. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In OSDI, pages 137–150, 2004.
6. J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. PVLDB, 4(11):1123–1134, 2011.
7. Z. Kaoudi and I. Manolescu. RDF in the clouds: a survey. VLDB J., 24(1):67–91, 2015.
8. K. Lee and L. Liu. Scaling queries over big RDF graphs with semantic hash partitioning. PVLDB, 6(14):1894–1905, 2013.
9. T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. VLDB J., 19(1):91–113, 2010.
10. K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store. In PSI EtA, page 4, 2010.
11. M. Stonebraker, D. J. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? Commun. ACM, 53(1):64–71, 2010.
12. P. Tsialiamanis, L. Siderirgos, I. Fundulaki, V. Christophides, and P. A. Boncz. Heuristics-based query optimisation for sparql. In EDBT, pages 324–335, 2012.
13. R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In First International Workshop on Graph Data Management Experiences and Systems, page 2. ACM, 2013.
14. R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In ACM Int’l Conf. on Management of Data SIGMOD, pages 13–24, 2013.
15. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI, pages 15–28, 2012.
16. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In 2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud’10, 2010.