# Efficiently Joining Group Patterns in SPARQL Queries

María-Esther Vidal[1] and Edna Ruckhaus[1] and Tomas Lampo[1] and Amadís Martínez[1,2] and Javier Sierra[1] and Axel Polleres[3]

[1] Universidad Simón Bolívar, Caracas, Venezuela
{mvidal,ruckhaus,tomas,amadis,javier}@ldc.usb.ve
[2] Universidad de Carabobo, Venezuela
aamartin@uc.edu.ve
[3] Digital Enterprise Research Institute
National University of Ireland, Galway, Ireland
axel.polleres@deri.org

**Abstract.** In SPARQL, conjunctive queries are expressed by using shared variables across sets of triple patterns, also called basic graph patterns. Based on this characterization, basic graph patterns in a SPARQL query can be partitioned into groups of acyclic patterns that share exactly one variable, or *star-shaped* groups. We observe that the number of triples in a group is proportional to the number of individuals that play the role of the subject or the object; however, depending on the degree of participation of the subject individuals in the properties, a group could be not much larger than a class or type to which the subject or object belongs. Thus, it may be significantly more efficient to independently evaluate each of the groups, and then merge the resulting sets, than linearly joining all triples in a basic graph pattern. Based on this observation, we have developed query optimization and evaluation techniques on *star-shaped* groups. We have conducted an empirical analysis on the benefits of the optimization and evaluation techniques in several SPARQL query engines. We observe that our proposed techniques are able to speed up query evaluation time for join queries with star-shaped patterns by at least one order of magnitude.

## 1 Introduction

In the context of RDF documents and SPARQL [1] queries, variables in (a combination of) basic graph patterns may be interpreted as either subjects characterized by a property, object values of a property, or properties that relate a subject and an object. These basic graph patterns in a query can be partitioned and reordered into groups of pattern combinations according to exactly one common variable, which we call *star-shaped* groups; the combinations may be over subjects, objects or mixed subjects and objects. For example, subject star-shaped groups of triple patterns around the same variable, commonly occur in complex SPARQL queries, where such groups around the same subject variable, typically describe a "class" of individuals to be queried: that is, a subject *star-shaped* group may be viewed as all the combinations of object values of the properties that characterize the individuals that belong to this "class".[4] Thus, the

[4] We quote "class" to indicate that we are not talking about classes in the sense of OWL or RDFS, but rather groups of individuals characterized by common properties.

number of triples in the group is proportional to the number of individuals in the subject; however, depending on the degree of participation of the subject individuals in the properties, the group may be not much larger than this subject "class". In the common case where *star-shaped* groups denote instances of data such that the combined properties imply quasi functional dependencies, the group may be significantly smaller than any of the participating properties taken on its own. Similar properties hold for object and subject-object *star-shaped* groups. Therefore, it is often beneficial to evaluate such groups jointly.

In this work, we have developed a query optimization technique that provides the basis for an efficient evaluation of common SPARQL queries that can be rewritten as combinations of small *star-shaped* groups. In order to identify an optimal plan for a query, a randomized cost-based optimizer partitions the basic graph patterns that appear in the 'WHERE' clause of a query into small-sized *star-shaped* groups, and also explores different orderings within groups and between groups. We focus in this paper on conjunctive queries, i.e., queries consisting of a single basic graph pattern (BGP) [1, Section 5.1]. However, since BGPs are the basic building block for any other, more complex patterns, such as OPTIONAL, UNION, FILTER, and GRAPH patterns, obviously our method also proves valuable for efficiently evaluating subpatterns within more complex queries. In addition, to efficiently evaluate SPARQL queries, we have developed two different physical join operators for the SPARQL query language that support the evaluation of combinations of such groups: we propose the *njoin* and the *gjoin* operators. The njoin scans the triples of the first pattern, and loops on the second pattern for matching triples. The gjoin evaluates star-shaped groups and matches their results. We have implemented the gjoin operator in the Jena query engine, and have empirically studied the performance of the optimized queries in several RDF query engines: our own system OneQL[2], Jena [3], RDF-3X [4], Sesame [5] and GiaBATA [6]. We have observed that our techniques are able to speed up the query evaluation time – especially for queries where common star-shaped patterns can be found – by several orders of magnitude.

To summarize, the main contributions of this paper are the following:

– We define and have implemented different physical operators for combining patterns: (1) a naive operator njoin, that for each triple that satisfies the first pattern, loops on the second pattern for matching triples, and (2) a gjoin operator which evaluates the *star-shaped* groups jointly, and matches their results.
– We define sampling techniques to estimate the result size and cost of star-shaped group patterns, the size of each property, and the selectivity of their subjects and objects, which help us to identify the most promising star-shaped groups.
– Based on the sampled values, we establish cost metrics that reflect the number of RDF triples that need to be read in order to answer the query. Out cost model extends the model presented in [2], to estimate the evaluation cost and cardinality of the *star-shaped* group physical operator.
– We describe a randomized optimization strategy that identifies a cost-effective plan (wrt. our metrics) and it is based on the Simulated Annealing algorithm. The algorithm explores execution plans of any shape (bushy trees) in contrast with other optimization algorithms that explore a smaller portion, e.g., left-linear plans only.

Bushy trees need to be explored, as query plans of any shape may be generated when combining star-shaped groups.
- In our evaluation we provide an empirical analysis on the predictive capability of our cost model, and the benefits of the proposed evaluation techniques on different RDF query engines.

The remainder of the paper is structured as follows. We start with a motivating example in the following section. Our query evaluation engine along with its underlying cost model and the query optimization strategy are presented in section 3. An extensive experimental study is reported in section 4, and section 5 summarizes the related work. Finally, we conclude in section 6 with an outlook to future work.

## 2 Motivating Example

As a running example throughout this paper we consider an RDF dataset on US Congress vote results, published as RDF at the *http://www.govtrack.us* website.[5] This dataset registers individuals and property values related to the US bills voting process grouped per year. E.g., for each of the 216 bills voted in 2004, it registers its title, date, voting options and winners; each voter and vote are also registered (there are 100 voters for each election). Table 1 reports the number of triples, and the number of different values for the subject and the object of some of the properties in the 2004 dataset.

| **property** | **# triples** | **# subject values** | **# object values** |
|---|---|---|---|
| voter | 21,600 | 21,600 | 100 |
| winner | 216 | 216 | 2 |
| hasBallot | 21,600 | 216 | 21,600 |
| option | 21,600 | 21,600 | 3 |
| title | 216 | 216 | 216 |

**Table 1.** Cardinality and number of values govtrack.us 2004

An example query posed against this dataset is *All the bills and their titles where 'Nay' was the winner, and at least one voter voted for the same option (Aye/Nay/NoVote) as voter 'L000174'*. There may be several equivalent (w.r.t. the set of answers) query evaluation strategies or plans where we may apply regrouping or reordering of basic graph patterns. Two equivalent SPARQL queries can be seen in Figures 1(a) and 1(b), and their query plan trees are presented in Figures 2(a) and 2(b), respectively; we denote our njoin and gjoin operators, which we will detail later, by $\bowtie^{njoin}$ and $\bowtie^{gjoin}$, respectively. Intuitively, the gjoin operator always applies when two groups of more than one triple pattern are joined, as opposed to joining a single triple with a group. The tree in Figure 2(a) is left-linear, whereas the tree in Figure 2(b) is a bushy tree where the patterns were partitioned into star-shaped groups.

[5] *http://www.govtrack.us/data/rdf/*

```
PREFIX vote: <tag:govshare.info,2005:rdf/vote/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX people:<http://www.rdfabout.com/rdf/usgov/>
SELECT ?E ?T
FROM <http://example.org/votes>
WHERE
        {?E vote:winner 'Nay' .
        ?E dc:title ?T .
        ?E vote:hasBallot ?I .
        ?I vote:option ?X .
        ?J vote:option ?X .
        ?E vote:hasBallot ?J .
        ?J vote:voter 'people:L000174'.
        FILTER (?I != ?J) }
```

(a) SPARQL Query without groupings

```
PREFIX vote: <tag:govshare.info,2005:rdf/vote/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX people:<http://www.rdfabout.com/rdf/usgov/>
SELECT ?E ?T
FROM <http://example.org/votes>
WHERE
        {{{?E vote:winner Nay .
        ?E dc:title ?T} .
        {?E vote:hasBallot ?I .
        ?I vote:option ?X}} .
        {?J vote:voter people:L000174 .
        ?J vote:option ?X .
        ?E vote:hasBallot ?J}.
        FILTER (?I != ?J)}
```

(b) SPARQL Query with groupings

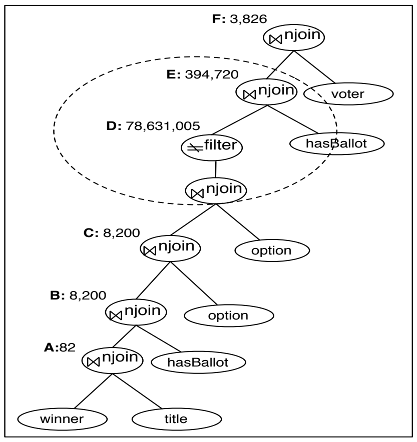
**Fig. 1.** Two Equivalent Queries

In an RDF dataset, there is only asserted knowledge, so the evaluation cost is proportional to the number of RDF triples that are read along query evaluation (i.e., the total number of intermediate results).

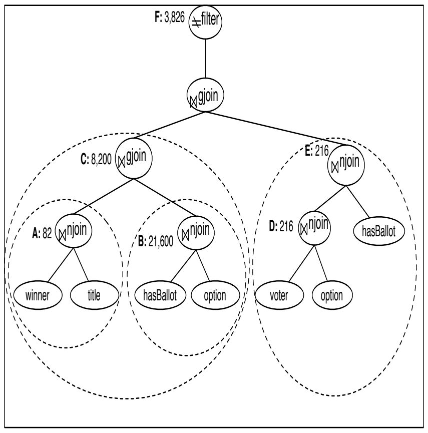
Let us take a look at the evaluation of the patterns. In plan 2(a), patterns are evaluated in a left linear fashion: for each triple in a pattern, we loop on the next pattern and retrieve the matching triples; this procedure continues until all the patterns are evaluated. Each sub-tree is annotated with its evaluation cost defined in terms of the number of RDF triples that are produced during execution time; the total query evaluation cost is 79,046,033 triples.

In plan 2(b), patterns were partitioned into three star-shaped groups. Similarly, each sub-tree is annotated with its evaluation cost, and some of the sub-trees correspond to the gjoin operator. The estimated cost is dramatically reduced to 34,140 triples; note that the ordering within each group is also relevant, e.g., if the instantiated pattern {?J vote:voter people:L000174} were the right-most pattern in the group instead of the left-most one, the total number of sub-tree **E** intermediate results, would have been larger. This example illustrates how the combination of appropriate partitioning in star-shaped groups and join reordering may improve the efficiency of query evaluation. It is beneficial to identify small-sized star-shaped groups and to avoid large-sized ones.

For instance, the small-sized group in sub-tree **D**, in Figure 2(b), reduces the number of voter options from 21,600 to 216 due to the voter instantiation {?J vote:voter people:L000174}. On the other hand, avoiding large-sized star-shaped groups prevents the explosion of the evaluation cost, e.g., the star-shaped group {?I vote:option ?X. ?J vote:option ?X} should be not be considered since the join selectivity of the 'option' object values is low, and a large number of matchings will occur and be accounted as intermediate answers. Note that in Figure 2(a), the cost of this plan explodes when the two 'option' property patterns are joined in sequence; this sequential join is marked with a dashed circle in Figure 2(a).



(a) Left-linear tree



(b) Bushy tree

**Fig. 2.** Query Execution Trees

## 3 Our Approach

In this section we define *star-shaped* groups and our proposed physical join operators. Moreover, we will elaborate on a cost model that describes the evaluation cost of these operators, and a cost-based query plan optimizer that is able to explore execution plans that combine *star-shaped* groups.

### 3.1 Star-shaped Group-based Query Engine

Formally, a star-shaped group is defined as follows:

**Definition 1** ($?X^*$**-BGP).**

*Each triple pattern* {*?X p o* } *or* {*s p ?X* } *such that* $s \neq ?X$*,* $p \neq ?X$*, and* $o \neq ?X$ *is a* Star-shaped basic graph pattern w.r.t. ?X*, or* $?X^*$*-BGP. If* $P$ *and* $P'$ *are* $?X^*$*-BGPs such that* $var(P) \cap var(P') = \{?X\}$ *then* $P \cup P'$ *is an* $?X^*$*-BGP.*

We have developed two physical operators for the evaluation of SPARQL's logical Join operator, sometimes denoted AND in the literature [7]. The njoin scans the triples of the first pattern, and loops on the second pattern for matching triples. The gjoin independently evaluates the star-shaped groups, and matches their results. These operators are described using the terminology presented in [7] for the semantics of SPARQL graph pattern expressions:

– A *triple pattern* (or a *basic graph pattern*, respectively) is an RDF triple (or a set of RDF triples, resp.) where variables may occur in subject, predicate or object position.
– Solutions of patterns are described in terms of *mappings* $\mu$ which are partial functions from variables to RDF terms.

– The evaluation of a basic graph pattern $P$, $[[P]]_G$ is the set of all mappings $\mu$ where $dom(\mu)$ is the set of variables occurring in $P$, such that $\mu(P) \subseteq G$. Each mapping corresponds to a valid pattern instantiation in graph $G$.
– *Compatible mappings* $\mu_1$, $\mu_2$ are those which coincide in the variables they share, and denote matching pattern instantiations.
– If $P$ is a pattern and $\mu$ is a mapping, then by $\mu \mid_P$ we denote the mapping obtained from restricting $\mu$ to the variables in $P$.

A detailed description of the operators is as follows:

– Nested-Loop Join $\bowtie^{njoin}$: Given two star-shaped group patterns $R_1$ and $R_2$, we extend each mapping $\mu_{R_1} \in [[R_1]]_G$ to $\mu'_{R_1} \supseteq \mu_{R_1}$ such that $\mu_{R_2} = \mu'_{R_1} \mid_{R_2}$ is in $[[R_2]]_G$. I.e., $\mu_{R_1}$ and $\mu_{R_2}$ are compatible mappings.
– Group Join $\bowtie^{gjoin}$: Given two star-shaped groups $R_1$ and $R_2$, each of them is independently evaluated, and the results are combined to match the compatible mappings.

## 3.2 Star-shaped Group Cost Model

In this section we define the cost model used for query plan generation. First, we describe the sampling technique used by the star-shaped cost model to estimate costs of intermediate results. Then, we present the application of this sampling technique for gathering RDF statistics, and for estimating the cost and cardinality of star-shaped groups. Finally, we define the formulas used for computing the cardinality, and the cost of a SPARQL query plan.

**Adaptive Sampling Techniques.** Unlike traditional approaches [8], the use of adaptive sampling techniques for query size and cost estimation does not require to store full summary statistics about the data. An additional advantage of this approach is that no strong assumptions about data characteristics are made upfront, but rather these properties can be estimated dynamically anytime. In [9], an adaptive sampling algorithm for estimating the result size of general queries is presented. It is applicable to any query that can be partitioned into disjoint sets of answers. This technique assumes that there is a population $\mathbb{P}$ of all the different valid instantiations of a predicate $P$, and that $\mathbb{P}$ is divided into $n$ partitions according to the instantiations of one or more arguments of $P$. Each element in $\mathbb{P}$ is related to its evaluation cost and cardinality, and the population $\mathbb{P}$ is characterized by the statistics mean and variance of these two parameters.

The objective of the sampling is to identify a sample of the population $\mathbb{P}$, called $\mathbb{EP}$, such that the mean and variance of the cardinality (resp., evaluation cost) of $\mathbb{EP}$ are valid to within a predetermined accuracy and confidence level.

To estimate the mean of the cardinality (resp., cost) of $\mathbb{EP}$, say $\overline{Y}$, within $\frac{\overline{Y}}{d}$ with probability $p$, where $0 \leq p < 1$ and $d > 0$, and $\alpha = \frac{d \times (d+1)}{(1-\sqrt{p})}$, the sampling method assumes an *urn* model.

The urn has $n$ balls from which $m$ samplings are repeatedly taken, until the sum $z$ of the cardinalities (resp., costs) of the samples is greater than $\alpha \times (\frac{S}{Y})$. The estimated mean of the cardinality (resp., cost) is: $\overline{Y} = \frac{z}{m}$

The values $d$ and $\frac{1}{(1-\sqrt{p})}$ are associated with the relative error and the confidence level, and $S$ and $Y$ represent the cardinality (resp., cost) variance and mean of $\mathbb{P}$. Since statistics of $\mathbb{P}$ are unknown, the upper bound $\alpha \times \frac{S}{Y}$ is replaced by $\alpha \times b(n)$.

To approximate $b(n)$ for cost and cardinality estimates, $k$ samples are randomly evaluated and the maximum value is taken among them:

$b(n) = max_{i=1}^{k}(card(P_i))$ (resp. $b(n) = max_{i=1}^{k}(cost(P_i))$, where $1 \leq k \leq n$

**Cardinality and Cost of Query plans** The cost of query plans and sub-plans is either computed or estimated according to their shape as follows:

- The adaptive sampling method is used to estimate the size and cost of star-shaped groups. The sampling process for groups is simple and gives accurate estimates: the population for the cost and cardinality estimates is comprised of the set of mappings $\mu$ in the evaluation of the first pattern in the group. We sample on this population, and for each sampled $\mu$, we evaluate the group and compute its cardinality (number of answers) and the evaluation cost. The samples are averaged in order to compute the final values for cost and cardinality.
- The cost of plans which are not star-shaped is computed according to cost formulas similar to the ones used in relational databases [8, 10].

## 3.3 Star-shaped Group Query Optimizer

The star-shaped group optimizer is implemented as a Simulated Annealing randomized algorithm which performs random walks over the search space of bushy query execution plans. Random walks are performed in stages, where each stage consists of an initial *plan generation step* followed by one or more *plan transformation steps*. An equilibrium condition or a number of iterations determines the number of transformation steps. At the beginning of each stage, a query execution plan is randomly created in the plan generation step. Then, successive *plan transformations* are applied to the query execution plan in order to obtain new plans. The probability of transforming a current plan $p$ into a new plan $p'$ is specified by an acceptance probability function $P(p, p', T)$ that depends on a global time-varying parameter $T$ called the *temperature*; it reflects the number of stages to be executed. The function $P$ may be nonzero when $cost(p') > cost(p)$, meaning that the optimizer can produce a new plan even when it is worse than the current one, i.e., it has a higher cost. This feature prevents the optimizer from becoming stuck in a local minimum. Temperature $T$ is decreased during each stage and the optimizer concludes when $T = 0$. Transformations applied to the plan during the random walks correspond to the SPARQL axioms of the physical operators implemented by the query and reasoning engine. The transformation rules that implement the axioms that define the $\bowtie^{njoin}$ and $\bowtie^{gjoin}$ operators are as follows:

1. Symmetry:
   - $R_1 \bowtie^{njoin} R_2 \equiv R_2 \bowtie^{njoin} R_1$
   - $R_1 \bowtie^{gjoin} R_2 \equiv R_2 \bowtie^{gjoin} R_1$
2. Associativity:

  - $(R_1 \bowtie^{njoin} R_2) \bowtie^{njoin} R_3 \equiv R_1 \bowtie^{njoin} (R_2 \bowtie^{njoin} R_3)$
  - $(R_1 \bowtie^{gjoin} R_2) \bowtie^{gjoin} R_3 \equiv R_1 \bowtie^{gjoin} (R_2 \bowtie^{gjoin} R_3)$
3. Distributivity (Linear to Bushy)
  - $(R_1 \bowtie^{njoin} R_2) \bowtie^{njoin} R_3 \equiv (R_1 \bowtie^{njoin} R_3) \bowtie^{gjoin} (R_2 \bowtie^{njoin} R_3)$
4. Grouping
  - $(R_1 \bowtie^{njoin} R_2) \bowtie^{njoin} (R_3 \bowtie^{njoin} R_4) \equiv (R_1 \bowtie^{njoin} R_2) \bowtie^{gjoin} (R_3 \bowtie^{njoin} R_4)$
5. Fold into a star-shaped group: $P_1$ and $P_2$ are $?X^*$-BGPs such that $var(P_1) \cap var(P_2) = \{?X\}$, then:
  - $P_1 \bowtie^{njoin} P_2 \Rightarrow (P_1 \bowtie^{njoin} P_2)$
6. Unfold a star-shaped group: $P_1$ and $P_2$ are $?X^*$-BGPs such that $var(P_1) \cap var(P_2) = \{?X\}$, then:
  - $(P_1 \bowtie^{njoin} P_2) \Rightarrow P_1 \bowtie^{njoin} P_2$

For each iteration in the inner loop of the optimization algorithm, a transformation rule is applied with a random probability. We have assigned a probability value to each transformation rule. To illustrate this, in the running example from Figure 2 the following rules have been applied to transform the left linear plan into the bushy plan:

1. Associativity
   $((P_1 \bowtie^{njoin} P_2) \bowtie^{njoin} P_3) \bowtie^{njoin} P_4 \equiv (P_1 \bowtie^{njoin} P_2) \bowtie^{njoin} (P_3 \bowtie^{njoin} P_4)$
2. Grouping
   $(P_1 \bowtie^{njoin} P_2) \bowtie^{njoin} (P_3 \bowtie^{njoin} P_4) \equiv (P_1 \bowtie^{njoin} P_2) \bowtie^{gjoin} (P_3 \bowtie^{njoin} P_4)$

## 4 Related Work

In the context of the Semantic Web, several query engines have been developed to access RDF documents efficiently [3, 4, 6, 11–14]. Jena [3] provides a programmatic environment for SPARQL, and it includes the ARQ query engine and indices which provide an efficient access to large datasets. The ARQ-Optimizer is a system that implements heuristics for selectivity-based Basic Graph Pattern optimization, proposed by Stocker et al. [15]. These heuristics range from simple triple pattern variable counting to more sophisticated selectivity estimation techniques; the optimization process is based on a greedy optimization algorithm which may explore a reduced portion of the space of possible plans, i.e., only left linear plans. Hence, query plans generated by the ARQ-Optimizer can sometimes be far from the optimal plans. The Jena Tuple Database or TDB [13] is a persistent graph storage layer for Jena. TDB works with the Jena SPARQL query engine (ARQ) to support SPARQL together with a number of extensions (e.g., property functions, aggregates, arbitrary length property paths).

Sesame [14] is an open source Java framework for storing and querying RDF data. It supports both SPARQL and SeRQL queries which are translated to Prolog; the join operator is implemented as sideways-passing of variable bindings, which is similar to our Nested Loop Join (njoin) operator.

RDF-3X [4] focuses on an index system, and its optimization techniques were developed to explore the space of plans that benefit from these index structures. The RDF-3X query optimizer implements a dynamic programming-based algorithm for plan enumeration, which imposes restrictions on the size of queries that can be optimized and evaluated. Indeed, in certain cases, these index-based plans could coincide with our optimized plans; however, the RDF-3X optimization strategies are not tailored to identify any type of bushy plans or to scale up to queries with at least one Cartesian product.

GiaBATA [6], a SPARQL engine built on top of the dlvhex reasoning engine for HEX-programs, and the DLVDB [16] ASP solver with persistent storage. DLVDB is an extension of DLV which provides interfaces with external databases, takes advantage of the optimization techniques implemented in the current DBMSs for improving reasoning efficiency. Weiss et al. [17] propose a main memory indexing technique that uses the triple nature of RDF as an asset. Two other approaches [18, 19] define two secondary memory index-based representations and evaluation techniques for RDF-based queries. Several different RDF store schemas have been proposed [20–22] to efficiently implement an RDF management systems on top of a relational database system. These approaches empirically show that a physical implementation of vertically partitioned RDF tables may outperform the traditional physical schema of RDF tables. Similarly to some of the existing state-of-the-art RDF systems, the optimization techniques are not tailored to identify and evaluate small-sized star-shaped groups.

Finally, we have implemented the star-shaped based optimization and evaluation techniques in our own system, OneQL [10], and we have empirically shown the benefits of these techniques in queries against medium-size datasets. However, because OneQL is implemented in Prolog, it is not able to scale up to very large datasets. To overcome this limitation, we have implemented the star-shaped operators in several of the abovementioned state-of-the-art RDF engines.

## 5 Experimental Results

We conducted an experimental study to empirically analyze the effectiveness of the proposed optimization and evaluation techniques in our own and several existing RDF engines. We report on the evaluation time performance of bushy plans comprised of star-shaped groups and identified by our proposed query optimizer. We compare the performance of the RDF query engines OneQL, Sesame, DLVDB, Jena TDB, and the extensions of Jena and RDF-3X that implement the gjoin operator, i.e., the $\bowtie^{gjoin}$.

**Dataset and Query Benchmark:** we use the real-world dataset on US Congress vote results of the 2004 bills voting process described in Table 1; the total size is 3.613 MB and 67,392 triples. We considered two sets of queries; benchmark one is comprised of 17 queries which are described in Figure 3(a) in terms of the number of patterns in the WHERE clause and the answer size; all the queries have at least one pattern whose object is instantiated with a constant. Benchmark two is a set of 120 queries which are composed of between 1 and 7 gjoin(s) among patterns of very small size. We also use the real-world ontology YAGO [23];[6] the total size of the dataset is 4GB and 44 millions of triples. We consider a benchmark of 10 queries which are comprised of between 17 and 25 basic patterns; for all these queries the answer is empty, except q6 that produces 551 triples. These three benchmarks are published in http:www.ldc.usb.ve/˜mvidal/OneQL/datasets.

[6] Ontology available for download at http://www.mpi-inf.mpg.de/yago-naga/yago/

**Evaluation Metrics:** we report on runtime performance, which corresponds to the *user time* produced by the *time* command of the Unix operation system. The experiments on dataset one were evaluated on a Solaris machine with a Sparcv9 1281 MHz processor and 16GB RAM; experiments on the dataset Yago and the RDF-3X were executed on a Linux Ubuntu machine with an Intel Pentium Core2 Duo 3.0 GHz and 8GB RAM. Jena extensions were developed in Java (64-bit JDK version 1.5.0_12); OneQL is implemented in SWI-Prolog (Multi-threaded, 64 bits, Version 5.6.54); finally, RDF-3X 0.3.3 is implemented in gcc/g++ v4.3.3.

**Query Engine Implementations:** the *star-shaped* group randomized query optimizer implements a Simulated Annealing algorithm that was run for twenty iterations and an initial temperature of 700; transformation rules were applied according to the probability distribution reported in Table 3(b).

To implement the gjoin operator in Jena 2.3, we modified the method `QueryIterator stream` in the class `com.hp.hpl.jena.sparql.engine.main.OpCompiler`. The Jena *GJoin* performs as follows: first, two objects of the `QueryIteratorCaching` class are created by replicating the input of the method `stream`; the outer and inner star-shaped groups are independently evaluated by calling the `compileOp` method with these `QueryIteratorCaching` objects. Each result set is stored in a different `QueryIterator` object, and the method `QueryIterJoin` is called to compute the join matches between the two sets. A new `QueryIterator` object is created to store the matches and is returned as the output of the method. We call this version of Jena, GJena.

Additionally, we have extended RDF-3X 0.3.3 to respect star-shaped plans produced by our query optimizer; njoin and gjoin are implemented as RDF-3X *Hash Join*s, while njoins in star-shaped groups correspond to RDF-3X *Merge Join*s; we call this version GRDF-3X. Finally, the njoin and gjoin operators were also implemented in DVLDB; for each star-shaped group in a plan, we generate a relational view which only projects the join variables of the star-shaped group.

| query | #patterns | answer size |
|---|---|---|
| q1 | 4 | 3 |
| q2 | 3 | 14,033 |
| q3 | 7 | 3,908 |
| q4 | 4 | 0 |
| q5 | 4 | 0 |
| q6 | 4 | 47 |
| q7 | 3 | 6,600 |
| q8 | 3 | 963 |
| q9 | 7 | 13,177 |
| q10 | 9 | 6,003 |
| q11 | 9 | 150 |
| q12 | 9 | 0 |
| q13 | 3 | 100 |
| q14 | 3 | 100 |
| q15 | 3 | 1 |
| q16 | 4 | 0 |

(a) Benchmark One

| Transformation Rule | Probability |
|---|---|
| Symmetry | 0.9 |
| Associativity | 0.9 |
| Linear To Bushy | 0.9 |
| Grouping (Folding into a star-shaped group) | 0.7 |
| Grouping (Unfold a star-shaped group) | 0.3 |

(b) Transformation Rules Distribution

**Fig. 3.** Experiment Configuration Set-Up

### 5.1 Effectiveness of the Star-shaped Group-Based Optimization Techniques

We study the effectiveness of the star-shaped group-based optimization techniques by empirically analyzing the quality of the optimized plans w.r.t. the rest of the plans of the corresponding queries, and the runtime performance of the optimized plans.

To analyze the quality of the optimized plans, we generated all the plans of q13, q14, q15, and q16 in Figure 3(a), and computed the percentile in which the optimal plan falls. This optimal plan was identified by the star-shaped group randomized optimizer, and all the plans were run on the version of Jena that implements the gjoin operator. Queries q13, q14, q15, and q16 fall in the 83th, 26th, 92th, and 99th percentiles, respectively. These results suggest that the optimizer is able to traverse the space of star-shaped groups and identify those that minimize the evaluation cost.

Table 2 compares the runtime cost of the non-optimized versus optimized versions of the first nine queries of benchmark one in the engines: DLVDB, Sesame, and Jena TDB (all versions).

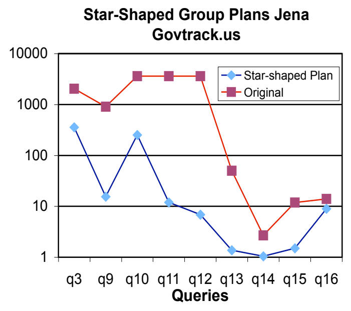
| Query | Jena TDB Fixed | Jena TDB Stats | Jena TDB None | Sesame | DLVDB |
|---|---|---|---|---|---|
| q1 | 0m6.952s | 0m7.149s | 0m6.861s | 0m0.34s | 0m0.37s |
| $q1_o$ | 0m4.072s | 0m4.320s | 0m4.048s | 0m0.03s | 0m0.34s |
| q2 | 0m29.106s | 0m29.443s | 0m28.553s | 0m1.30s | 0m1.23s |
| $q2_o$ | 0m28.404s | 0m31.121s | 0m29.308s | 0m1.38s | 0m1.16s |
| q3 | 129m56.301s | 150m7.661s | 133m11.581s | Timeout(100s) | 14m53.05s |
| $q3_o$ | 0m19.945s | 0m20.816s | 0m20.615s | 0m0.56s | 0m10.614s |
| q4 | 0m32.681s | 0m32.364s | 0m31.928s | 0m2.45s | 45m47.620s |
| $q4_o$ | 2m43.325s | 2m33.943s | 2m37.936s | 0m52.18s | 0m9.189s |
| q5 | 0m26.085s | 0m26.095s | 0m26.187s | 0m1.71s | 33m1.786s |
| $q5_o$ | 0m4.114s | 0m4.097s | 0m3.902s | 0m1.06s | 22m14.712s |
| q6 | 0m7.958s | 0m7.808s | 0m7.691s | 0m0.26s | 0m0.745s |
| $q6_o$ | 0m4.939s | 0m4.951s | 0m4.772s | 0m0.02s | 0m0.712s |
| q7 | 0m20.486s | 0m20.736s | 0m20.342s | 0m1.01s | 0m0.761s |
| $q7_o$ | 0m20.376s | 0m20.381s | 0m19.733s | 0m0.56s | 0m0.699s |
| q8 | 0m12.802s | 0m12.758s | 0m13.033s | 0m0.28s | 0m0.339s |
| $q8_o$ | 0m0.402s | 0m11.181s | 0m11.677s | 0m0.09s | 0m0.314s |
| q9 | 337m12.276s | 333m54.96s | 350m14.346s | Timeout (100s) | 1191m55.86s |
| $q9_o$ | 0m12.238s | 0m11.689s | 0m11.660s | 0m0.58s | 415m43.86s |

**Table 2.** Evaluation Time of Non-Optimized versus Optimized Queries (seconds)

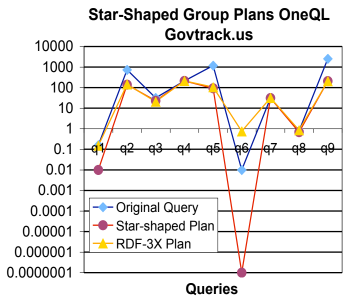
We can observe that in general, the optimization techniques are able to speed up the evaluation time in almost all the SPARQL query engines. Particularly, in query q3, the improvement was produced by the reordering of the patterns, while in query q9, a combination of reordering and star-shaped groups was generated. However, the randomized optimizer was not able to produce the star-shaped groups which would have improved the evaluation time of q4.

Additionally, we study the effectiveness of the star-shaped group-based optimization techniques in different Jena engines. We ran queries q3, q9, q11, q12, q13, q14, q15, and q16 on Jena 2.3 and their respective optimized plans in GJena, and compare their evaluation times (seconds-logarithmic scale) in Figure 4(a). All these plans were composed of small-sized star-shaped groups. We could observe that the evaluation time of the optimized queries was at most 50% of the evaluation time of Jena's native query processing, and in some cases, the time was reduced by up to two orders of magnitude. These results indicate that by identifying groups our optimizer is able to generate significantly better plans for star-shaped queries compared to the "flat" join-reordering techniques in Jena.

Furthermore, we studied the benefits of the star-shaped based optimization and evaluation techniques by empirically analyzing the quality of star-shaped optimized plans w.r.t. the plans optimized by the RDF-3X query optimizer. Nine queries of benchmark one were optimized by OneQL and RDF-3X, and the generated plans were run in OneQL. Each RDF-3X optimized plan was run using the gjoin to evaluate the groups in the bushy plans, and using the njoin to evaluate joins inside the star-shaped groups. Figure 4(b) reports the evaluation time (seconds-logarithmic scale) of these combinations of queries. We can observe that the evaluation time of the star-shaped and RDF-3X optimized plans are competitive, except for queries *q1* and *q6* where our optimizer was able to identify plans where all the triples are instantiated, and the most selective ones are evaluated first. These results indicate that the star-shaped based optimization and evaluation techniques may be used in conjunction with the state-of-the-art techniques to provide more efficient query engines; they have encouraged us to develop our physical operators in existing RDF engines like Jena and RDF-3X.



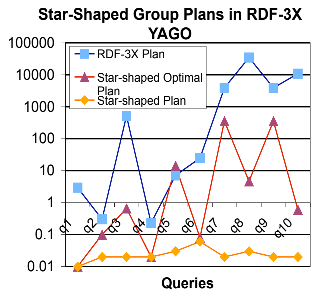
(a) Jena Evaluation Time



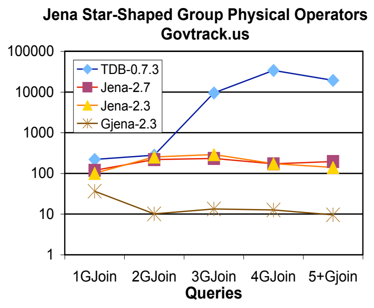
(b) OneQL Evaluation Time

**Fig. 4.** Effectiveness of the Star-shaped Optimizer-Time (seconds-logarithmic scale)

Finally, we studied the behavior of the star-shaped plans in RDF-3X against large datasets. For each query of benchmark three on YAGO, we computed the RDF-3X optimal plan and the star-shaped plan produced by OneQL. We also built the optimal star-shaped group plan of the query *by hand*; each optimal plan was comprised of between two and five star-shaped groups free of Cartesian products. RDF-3X optimized plans were run in RDF-3X, while the other two versions were executed in GRDF-3X. Figure 5(a) reports on the evaluation time (seconds-logarithmic scale) of these queries; optimization time is not considered in any case. We can observe that the star-shaped plans produced by our optimizer can reduce evaluation time by up to three orders of magnitude, and in many cases their evaluation time is close to the optimal cost. We also ran this experiment in Jena; original queries were run in Jena 2.3 and the other two versions were executed in GJena. We observe a similar trend in the evaluation time.



(a) RDF-3X Evaluation Time



(b) Jena Engines Evaluation Time

**Fig. 5.** Performance of Star-shaped Groups-Time (seconds-logarithmic scale)

### 5.2 Performance of the Star-shaped Group Physical Operators

We have conducted an empirical analysis on the benefits of the evaluation techniques implemented on Jena, and we have executed the one hundred and twenty queries of the benchmark two on govtrack.us. We compared the benefits of using our gjoin physical implementation and the njoin implementation provided by Jena versions 2.3, Jena 2.7 and Jena TDB. We could observe that our gjoin implementation was able to speed up the evaluation time by up to three orders of magnitude. Figure 5(b) reports the average time (seconds-logarithmic scale) consumed by the Jena query engine to evaluate the different queries.

## 6 Conclusions and Future Work

We have defined optimization and evaluation techniques that provide the basics for an efficient evaluation of SPARQL queries. The assumptions of the uniformity of values of subjects and objects in a property, and of independence between properties, may lead to imprecise estimates in real-world problems. Thus, in the future we plan to enhance the star-shaped cost model with Bayesian inference capabilities so as to consider the lack of uniformity of the values in the RDF documents, and correlations between patterns in the SPARQL queries. Furthermore, we plan to conduct an experimental comparison of the performance of our optimizer and the RDF-3X optimizer [24]; also, a detailed experimental study of column-store approaches[22] such as Virtuoso [7] or MonetDB[8] is on our agenda.

## 7 Acknowledgements

This research has been partially supported by the DID-USB and the Proyecto ALMA Mater-OPSU. The work of Axel Polleres is supported by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2). We thank Andy Seaborne for his advice on the installation of the different Jena engines.

## References

1. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Recommendation (January 2008) available at http://www.w3.org/TR/rdf-sparql-query/.
2. Ruckhaus, E., Ruiz, E., Vidal, M.: OneQL: An Ontology Efficient Query Language Engine for the Semantic Web. In: Proceedings ALPSWS2007. (2007)
3. Dickinson, I.: The JenaOntology Api. http://jena.sourceforge.net/ontology/index.html
4. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. PVLDB **1**(1) (2008) 647–659
5. Broekstra, J., Kampman, A., Harmelen, F.: Sesame: An Architecture for Storing and Querying RDF Data and Schema Information. In: Spinning the Semantic Web. (2003) 197–222
6. Ianni, G., Krennwallner, T., Martello, A., Polleres, A.: Dynamic Querying of Mass-Storage RDF Data with Rule-Based Entailment Regimes. In Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K., eds.: Proceedings of the 8th International Semantic Web Conference (ISWC 2009). Volume 5823 of LNCS., Washington DC, USA, Springer (October 2009) 310–327
7. Perez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: International Semantic Web Conference. (2006) 30–43
8. Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., Price, T.: Access Path Selection in a Relational Database Management System. Proceedings of ACM Sigmod (1979) 23–34
9. Lipton, R., Naughton, J.: Query size estimation by adaptive sampling (extended abstract). Proceedings of ACM Sigmod (1990) 40–46
10. Lampo, T., Ruckhaus, E., Sierra, J., Vidal, M.E., Martinez, A.: OneQL: An Ontology-based Architecture to Efficiently Query Resources on the Semantic Web. In: The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems at ISWC. (2009)
11. Franz Inc.: AllegroGraph. http://www.franz.com/agraph/allegrograph/
12. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: ISWC/ASWC. (2007) 211–224

[7] http://virtuoso.openlinksw.com/
[8] http://monetdb.cwi.nl/

13. Seaborne, A.: Jena TDB. http://openjena.org/wiki/TDB
14. Wielemaker, J.: An Optimised Semantic Web Query Language Implementation in Prolog. In: ICLP. (2005) 128–142
15. Stoker, M., Seaborne, A., Bernstein, A., Keifer, C., Reynolds, D.: SPARQL Basic Graph Pattern Optimizatin Using Selectivity Estimation. In: Proceedings of the 17th International Conference on World Wide Web (WWW). (2008) 595–604
16. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. Theory Pract. Log. Program. **8**(2) (2008) 129–165
17. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. PVLDB **1**(1) (2008) 1008–1019
18. Fletcher, G., Beck, P.: Scalable Indexing of RDF Graph for Efficient Join Processing. In: Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM). (2009) 1513–1516
19. McGlothlin, J., Khan, L.: RDFKB: Efficient Support For RDF Inference Queries and Knowledge Management. In: Proceedings of International Database Engineering and Applications Symposium (IDEAS). (2009) 259–266
20. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.: SW-Store: a vertically partitioned DBMS for Semantic Web data management. VLDB J. **18**(2) (2009) 385–406
21. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: Proceedings of the 33th International Conference on Very Large Data Bases (VLDB). (2007) 411–422
22. Sidirourgos, L., Goncalves, R., Kersten, M.L., Nes, N., Manegold, S.: Column-store support for RDF data management: not all swans are white. PVLDB **1**(2) (2008) 1553–1563
23. Suchanek, F.M., Kasneci, G., Weikum, G.: YAGO: A Large Ontology from Wikipedia and WordNet. Elsevier Journal of Web Semantics **6**(3) (2008) 203–217
24. Neumann, T., Weikum, G.: Scalable join processing on very large rdf graphs. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Providence, Rhode Island, USA, ACM Press (2009) 627–640