Exploiting the query structure for efficient join ordering in SPARQL queries

Andrey Gubichev
TU München
Germany
gubichev@in.tum.de

Thomas Neumann
TU München
Germany
neumann@in.tum.de

ABSTRACT

The join ordering problem is a fundamental challenge that has to be solved by any query optimizer. Since the high-performance RDF systems are often implemented as triple stores (i.e., they represent RDF data as a single table with three attributes, at least conceptually), the query optimization strategies employed by such systems are often adopted from relational query optimization. In this paper we show that the techniques borrowed from traditional SQL query optimization (such as Dynamic Programming algorithm or greedy heuristics) are not immediately capable of handling large SPARQL queries. We introduce a new join ordering algorithm that performs a SPARQL-tailored query simplification. Furthermore, we present a novel RDF statistical synopsis that accurately estimates cardinalities in large SPARQL queries. Our experiments show that this algorithm is highly superior to the state-of-the-art SPARQL optimization approaches, including the RDF-3X's original Dynamic Programming strategy.

1. INTRODUCTION

The not-so-recent interest in the RDF data model and its query language SPARQL has already led to the development of several academic and commercial systems for storing and querying large RDF datasets. The common way to store linked data – employed by the *triple stores* – is to conceptually view RDF data as a single table with three columns that correspond to *Subject*, *Predicate*, *Object*. Starting with Hexastore [19] and RDF-3X [13], the triple stores have become a de-facto research and industry standard in RDF storage, with the state-of-the-art commercial triple stores being able to index up to 1 Trillion triples[1] (e.g., Virtuoso[20], AllegroGraph [1], BigOWLIM [4] and others) At the same time, the available RDF datasets are both increasing in size and in quality, that is having better structure. As the quality of the data improves, users tend to ask interactive queries of increasing complexity, just like it is commonplace with SQL in modern RDBMS's. For example, the query log of the interactive DBpedia endpoint has SPARQL queries with up to 10 joins [3], and analytical queries in the biomedical domain can include more than 50 joins [15].

[1] *http://www.w3.org/wiki/LargeTripleStores*

(c) 2014, Copyright is with the authors. Published in Proc. 17th International Conference on Extending Database Technology (EDBT), March 24-28, 2014, Athens, Greece: ISBN 978-3-89318065-3, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

Finding the right join order is known to be a challenging task for any relational optimizer. In RDF systems, additionally, the optimizers are faced with extremely large sizes of queries due to verbosity of the data format, and with the lack of schema that challenges the cardinality estimation process, an essential part of any cost-based query optimizer. It is easy to construct a query with less than 20 triple patterns whose compilation time (dominated by finding the optimal join order) in the high-performance RDF-3X system [13] is one order of magnitude higher than the actual execution time (see Appendix for an example of such query). On the other hand, another popular high-performance triple store, Virtuoso 7, seems to spend much less time finding the join order (probably employing some kind of greedy heuristics), but pays a high price for the (apparently) suboptimal ordering. For that specific query we have measured the following compile and runtimes in two systems:

System	Compile Time	Run Time
RDF-3X	78 s	2 s
Virtuoso	1.3 s	384 s

Ideally, we would like to have a hybrid of two approaches: a heuristics that spends a reasonable amount of time optimizing the query, and yet gets a decent join order.

This problem becomes even more pressing, as emerging applications require the execution of queries with 50+ triple patterns [15]. One of the popular alternatives for Dynamic Programming for such queries – Greedy heuristics – faces a hard challenge of greedily selecting even the first pair of triples to join due to structural correlations between different triple patterns[11]. Indeed, a triple pattern can be quite selective itself (e.g., *people born in France*), but not within the considered group of triple patterns (that could describe, e.g., *French Physicists*).

In this paper we propose a novel join ordering algorithm aimed at large SPARQL queries. Essentially, it is a SPARQL-tailored query simplification procedure, that decomposes the query's join graph into star-shaped subqueries and chain-shaped subqueries. For these subqueries we introduce a novel linear-time heuristics that takes into account the underlying data correlations to construct an execution plan. The simplified query graph typically has a much smaller size compared to the original query, thus allowing to run Dynamic Programming on it. In order to estimate the cardinalities in the simplified query, we introduce new statistical synopsis coined *Characteristic Pairs*. To validate the effectiveness and efficiency of our join ordering strategy, we run our join ordering algorithms on thousands of generated queries over real-world datasets, and compare them with the state-of-the-art SPARQL query optimization algorithms.

Note that, although we concentrate on triple stores in this paper, the problem of join ordering is orthogonal to the underlying physical storage and is therefore common to all the RDF systems. The

10.5441/002/edbt.2014.40

sources of this problem lie in verbosity of the RDF data model and high irregularity of real-world datasets. Our solutions (join ordering algorithms and statistical data structures for cardinality estimation) do not assume any particular organization of a triple store (i.e., specific indexes, data partitioning etc.) and are therefore applicable to a wide range of systems.

The rest of the paper is organized as follows. Sec. 2 introduces the concepts of cost-based SPARQL query optimization. Sec. 3 demonstrates the unique query optimization challenges caused by the nature of RDF data. Sec. 4 describes our novel algorithm for star-shaped SPARQL query optimization. Sec. 5 shows how this algorithm can be incorporated into the general SPARQL query optimizer, as well as presents a novel statistical synopsis for accurate cardinality estimations beyond star-shaped queries. Sec. 6 discusses related work. Sec. 7 evaluates the quality of our algorithms and other approaches.

2. PRELIMINARIES

In this section we will introduce the basic concepts and the state-of-the-art techniques of SPARQL query optimization.

2.1 Query Graph and SPARQL Graph

Given a SPARQL query, the query engine starts off with constructing its representation called a *query graph*. Specifically, every triple pattern of the query is turned into a node of the query graph, and two nodes are connected if the corresponding triple patterns share a common variable (or, if there is a FILTER condition relating variables of these two triple pattern). Conceptually, the nodes of the query graph entail scans of the dataset with the corresponding variable bindings, and the edges correspond to the join possibilities within the query. Thus defined, the query graph of a SPARQL query corresponds to the traditional query graph from relational query optimization, where nodes are relations and join predicates form edges.

Another way to represent a SPARQL query is a graph structure that we call a *SPARQL graph*. Its nodes denote variables of the query, while the triple patterns form edges between the nodes. Intuitively, this structure describes the subgraph that has to be extracted from the dataset.

Consider an example SPARQL query and its two representations (the SPARQL graph and the query graph) depicted in Figure 1. The query itself has 8 triple patterns (denoted $p_1 \dots p_8$). The SPARQL graph (Figure 1b) describes the pattern that has to be matched against the dataset: we are looking for two star-shaped patterns (around variables $?p$ and $?city$) that are connected via the two-hop chain $?p{-}?book{-}?city$. The equivalent query graph (Figure 1c) consists of two four-node cliques (induced by variables $?p$ and $?city$ that appear in four triple patterns each), and a chain between them.

2.2 Cost model

Given the query graph as input, the optimizer returns the query plan, which is defined by the *ordering of joins* between triple patterns, and the type of each join (merge or hash join). A cost-based query optimizer explores the search space of different algebraically equivalent query plans, and selects the optimal (cheapest) plan according to some cost function. Traditionally, the cost function takes into account the amount of intermediate results produced by joins in the query plan. For example, the RDF-3X cost functions for merge and hash joins (MJ and HJ, respectively) are defined as follows:

$$cost_{MJ} = \frac{lc + rc}{100}, cost_{HJ} = 300,000 + \frac{lc}{100} + \frac{rc}{10}, \quad (1)$$

where lc and rc are the cardinalities of the left and right inputs of the join operation (with $lc < rc$).

We note that even the exact join ordering algorithms (such as Dynamic Programming) yield the plan which is optimal only according to the *estimated* cost function. It is possible, however, that the actual cost function value is different from the estimated one (due to errors in intermediate result size estimations), and this could turn an estimated optimal plan into the de-facto suboptimal.

2.3 Cardinality estimation

In order to improve the cardinality estimates for star-shaped subqueries, Neumann and Moerkotte [11] suggested the data structure coined *characteristic set*. The characteristic set for a subject s is defined as $S_c(s) = \{p | \exists (s, p, o) \in \text{ dataset } R\}$. Essentially, the characteristic set for s defines the properties (attributes) of an entity s, thus defining its class (type) in a sense that the subjects that have the same characteristic set tend to be similar. For example, the class of *Person* in the knowledge base can be defined by the characteristic set $\{bornIn, livesIn, hasName, marriedTo, hasChild\}$. The authors of [11] note that in real-world datasets the number of different characteristic sets is surprisingly small (in order of few thousands), so it is possible to explicitly store all of them with their number of occurrences in the RDF graph.

As an example, consider again the query in Figure 1a and suppose that we are to estimate the cardinality of the join between the triple patterns containing *wrote* and *bornIn* predicates. The natural way to do it is to iterate over all characteristic sets, find those sets that contain both predicates, and sum up their counts. Note that this type of computation works for any number of joins within the same star subquery and delivers accurate estimates [11].

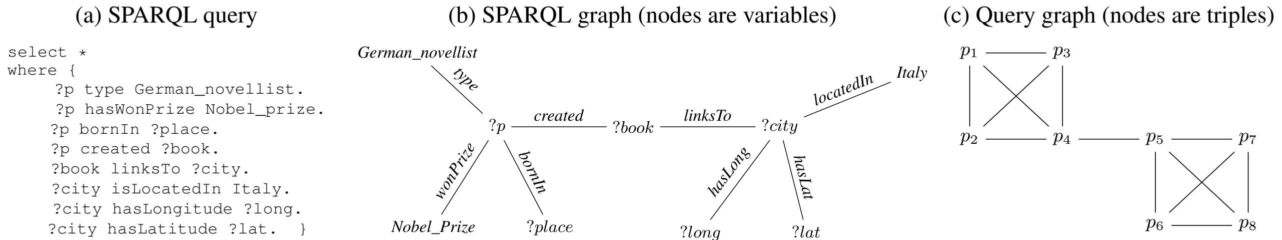
2.4 Physical Plan

A common approach towards storing RDF data is to index several (or even all possible) permutations of Subject, Predicate, Object in separate B^+-trees [16, 13, 18]. Since the data is available in several orderings, it is often beneficial to use a merge join as a physical implementation of a join operation. For example, the star subqueries typically are executed using the merge join. If we denote by σ_{order} the table scan in the given $order$ (e.g., Object-Predicate-Subject) with the specified constants, the star-shaped subquery around $?p$ from Figure 1b can be transformed into the following sequence of merge joins:

$$\Big(\Big(\sigma_{OPS}(O = \text{Nobel_prize}, P = \text{wonPrize}) \\ \bowtie_{MJ} \sigma_{OPS}(O = \text{German_novellist}, P = \text{type})\Big) \\ \bowtie_{MJ} \sigma_{PSO}(P = \text{wrote})\Big) \\ \bowtie_{MJ} \sigma_{PSO}(P = \text{bornIn})$$

Runtime techniques like *sideways information passing* stress the importance of merge joins even further [12]. Namely, the values of variable $?p$ encountered during the index scans can be propagated to other (less selective) index scans participating in the subquery with merge joins, such that the latter could skip most of their pages on disk. Propagation is possible since the merge join keeps the ordering of its input data intact.

Figure 1: Query graphs of a SPARQL query



While some systems explicitly maximize the number of merge joins during the plan generation [18], others do so by tuning the cost function (note, for example, the 300,000 cost unit penalty for a hash join in (1)).

3. CHALLENGES

Finding the optimal join order of a SPARQL query is very challenging due to the nature of RDF data. **First**, the RDF triple format is very verbose. Thus, for example, the TPC-H Query 2 written in SPARQL contains joins between 26 index scans (as opposed to joins between 5 tables in the SQL formulation!). The number of possible query plans is in the order of factorial of the table number, i.e. 5! = 120 plans in SQL vs 26! = $4 \cdot 10^{26}$ plans in SPARQL. In real applications it is not uncommon to see SPARQL queries with joins involving a hundred index scans (see Appendix of [15] for an example from the biomedical domain). This is a price that the engine has to pay for the flexibility of the data schema. **Second**, the lack of schema leaves the optimizer without essential information that is readily available to any relational optimizer, such as the set of tables, their attributes, primary and foreign keys. For example, a relational system would have information that an entity of a type *Person* has attributes *Name*, *Birthday*, *Birthplace* etc., and foreign key relationships to other entities (*Places*, *Companies* etc). The relational optimizer can therefore keep the statistics on these attributes and foreign keys (e.g., an average person has lived in 3 different places) and use it for result size estimation. All this information is only *implicitly* present in RDF data, where attributes and foreign keys become *structural correlations* in the RDF graph. In other words, some of the predicates tend to occur together as labels of outgoing edges of the same node (e.g., *bornIn*, *hasName* and *created*), and some subgraphs tend to occur together in RDF graphs (e.g.,writers and books).

The simplest of these correlations – corresponding to the attributes of the same entity – are captured by Characteristic Sets (CS) [11]. However, the DP algorithm requires computing the CS-based estimates for every non-empty subgraph of the query. It significantly increases the (already high) runtime of the DP algorithm, as we will demonstrate in the evaluation section. Moreover, CS do not capture correlations between different subgraphs in the RDF graph.

Together, these characteristics of RDF data create the following challenges for the query optimizer. First, the sheer size of the search space for large SPARQL queries does not allow the standard Dynamic Programming exploration, since it has to look at all the valid plans in order to find the cheapest one. Second, even for the mid-sized query graphs, the Dynamic Programming algorithm ignores the structure of the query, and therefore considers a lot of a priori suboptimal subplans during the plan construction. Third, the optimizer under the independence assumption fails to estimate the result sizes of most of the partial plans.

For instance, for the query in Figure 1a, the following entries are added to the DP table:

Partial Plan	Est. Result Size
$(?p, wrote, ?book) \bowtie (?book, linksTo, ?city)$	1.5 Mln
$(?book, linksTo, ?city) \bowtie (?city, hasLatitude, ?lat)$	1.3 Mln

Here, the independence assumption leads to a significant underestimation of the cost function, since the optimizer merely multiplies the frequencies of two predicates, thus getting a rather small join selectivity. In reality, however, books tend to link to multiple entities mentioned in them, so the selectivities of both joins are way higher than 1. The real cost of the partial plans is therefore much worse than what the optimizer expects. Indeed, the first subplan returns all the people that have written any book that links to any entity, and the second subplan yields all the entities linked to each other, such that one of them has a latitude as an attribute. Clearly, performing these chain-shaped joins earlier during query execution would produce enormous intermediate results. However, completely avoiding chain joins can not be made the 'rule of a thumb', since some of the chains yield extremely small results. The (nearly optimal) join ordering strategy is to split the query into star- and chain-subqueries while still keeping the correlations between different subqueries.

The contribution of this work, the Dynamic Programming-based *heuristics* overcomes these challenges as follows. It decomposes the SPARQL graph into the disjoint star-shaped subqueries and the chains connecting them. Having done that, we no longer need to consider joins between individual triple patterns of star- and chain-shaped subqueries (like in the table above) and thus drastically reduce the search space while keeping the plans very close to optimal. Furthermore, the plans for the star-shaped subqueries are found using the novel linear-time join ordering algorithm. This way, only the join possibilities between different subqueries contribute to the problem's exponential complexity, therefore reducing the search space size to the SQL level (e.g., down to 5! plans instead of 26! plans for the TPC-H query 2). To capture the correlations between different star subqueries, we introduce a generalization of characteristic sets (coined the *characteristic pairs*). This statistics helps estimating the cardinalities of joins between different stars, which in turn are used to order subqueries in the overall query plan.

4. STAR QUERIES OPTIMIZATION

In this section we describe our first contribution, a join ordering algorithm for star-shaped SPARQL queries. We start by introducing a statistical data structure coined the *hierarchical characterisation* of the RDF graph, and then describe the algorithm that employs it to find an (almost optimal) ordering of joins in star queries.

4.1 Hierarchical Characterisation

A characteristic set, defined in [11] as a set of outgoing edge labels for the given subject, tend to capture the semantic similarity between entities described in the RDF dataset. Thus, entities with the characteristic set {*hasTitle*, *hasAuthor*} usually describe books. While the lack of fixed schema prevents us from assigning these entities a type "Books" (e.g., we are not going to store such entities in a separate table), the characteristic sets allow us to predict selectivities for given sets of query triple patterns.

In real-world RDF datasets, set inclusion between different characteristic sets also bears some semantic information. Consider the following CS that describes a type *Writer*: $S_1 = \{hasName, bornIn, wroteBook\}$, and another CS that characterizes entities of type *Person*: $S_2 = \{hasName, bornIn\}$. Note that $S_2 \subseteq S_1$, and at the same time *Writer* is a subtype of *Person*. We find this situation to be extremely frequent in knowledge bases. For instance, the majority of characteristic sets in the Yago knowledge base [8] are subsets of each other, reflecting the class hierarchy of Yago entities.

Along with the set of predicates, the Characteristic Set (CS) keeps the number of occurrences of this predicate set in the dataset [11] (denoted as $count(CS)$). Here we introduce a generalization of this measure, namely the aggregate characteristic of CS $cost(CS)$. It is defined as the sum of occurrences of all the supersets of the given CS:

$$cost(CS) = \sum_{S \text{ is a char.set } \& \ CS \subseteq S} count(S)$$

The difference between $cost(CS)$ and $count(CS)$ is two-fold. First, $count(S)$ merely reflects the number of the star-shaped subgraphs of R that have those and only those edge labels mentioned in S. At the same time, $cost(S)$ is the number of subgraphs that have all the labels from S plus some other labels. In other words, $cost(S)$ for $S = \{p_1, \dots, p_k\}$ provides an estimate for the result size of the query

select distinct ?s
where $\{?s\ p_1\ ?o_1.\ \dots\ ?s\ p_k\ ?o_k\}$

Second, *cost* can be applied to any set of predicates that does not form the Characteristic Set. For example, the set of two predicates $S = \{hasName, wroteBook\}$ is not characteristic, since these two predicates are always accompanied by *bornIn*. However, $cost(S)$ still can be used to estimate the size of the join of two corresponding triple patterns, namely $(?s, hasName, ?name)$ with $(?s, wroteBook, ?book)$.

The following obvious property holds for two sets S_1, S_2 such that $S_2 \subseteq S_1$: $count(S_2) \geq count(S_1)$. For instance, the number of entities of type *Person* is clearly not smaller than the number of *Writers*. Same holds for the costs of sets in this situation: $cost(S_2) \geq cost(S_1)$.

At the same time, some subsets of the predicate set S may be cheaper than others. Consider again $S = \{hasName, bornIn, wroteBook\}$ and all its two-element subsets along with their costs listed in the table below:

Subset	is CS?	$cost$(Subset)
{*hasName*, *bornIn*}	yes	74K
{*hasName*, *wroteBook*}	no	43K
{*bornIn*, *wroteBook*}	no	39K

Notice that the last subset is the rarest occurring in the dataset (i.e., with the minimal cost), and it does not form the characteristic set. From the query optimizer's perspective, this means that, when given a query joining triples with these three predicates, the best strategy is to first join *bornIn* and *wroteBook* triple patterns, since the amount of intermediate results (i.e., *cost* of that two-element set) is the smallest. In order to make such kind of decisions possible, every characteristic set should have a pointer to its cheapest (in terms of *cost*) subset.

We capture these observations in the following formal definition. A *Hierarchical Characterisation* of the dataset R is the set $\{H_0, \dots, H_k\}$, such that

1. H_0 is the set of all characteristic sets of R
2. $H_i = \{ \underset{\forall C \subset S \wedge |C| = |S|-1}{\operatorname{argmin}} cost(C) \mid \forall S \in H_{i-1}\}$, that is H_i consists of the subsets C of sets from H_{i-1} that minimize $cost(C)$.
3. $\forall S \in H_k$: $|S| = 2$
4. every $S \in H_{i-1}$ stores a pointer to its cheapest subset $C \in H_i$.

Informally, the Hierarchical Characterisation of the dataset is a forest-like data structure of sets, where there is a link from S_1 to S_2, if S_2 is the cheapest subset of S_1 among all the subsets of S_1 that are just one element smaller than $|S_1|$.

An example of the part of the Hierarchical Characterisation for the Yago dataset [8] is given in Figure 2a. There, two characteristic sets in H_0 (IDs 195 and 154) point to the same cheapest subset from H_1.

As we have noted, in real RDF datasets most of characteristic sets are subsets of each other. This leads to the same sets appearing in different levels of the HC: both as leaves in the H_0 level, and as subsets of other sets in some H_i level. Naturally, we do not need to store the same set twice, so this ambiguity stays only on conceptual level. In theory, if H_0 level has m different sets drawn from n distinct elements, we can get up to $m \cdot n$ distinct subsets in HC. In practice, however, due to the fact that the same sets are shared between different levels, and different sets can get the same cheapest subsets, this number is quite close to m.

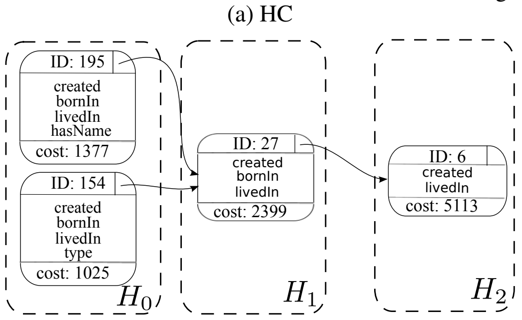
4.2 Computing Hierarchical Characterisation

A straightforward computation of Hierarchical Characterisation could be organized as follows: starting with Characteristic Sets, at each iteration for every set we find all its subsets (that are one element smaller), get the cheapest one, pass all the cheapest subsets for each set to the next iteration. The iterations should repeat until all the newly generated sets are of size 2. This however, generates the same set many times, since (a) the sets may appear in multiple levels of HC, (b) two different sets may have the same cheapest subset, as it is the case with sets 195 and 154 in Figure 2. An accurate theoretical analysis of this observation is out of scope of this paper.

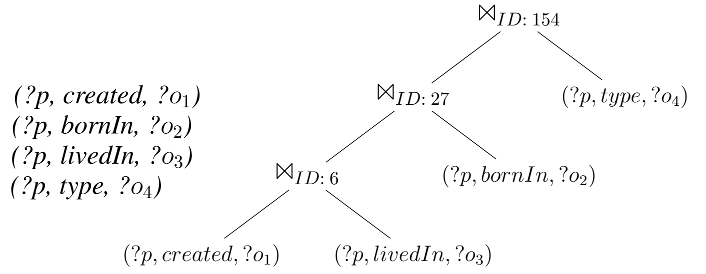
The computation works as follows. First, the Characteristic Sets are computed [11]. They are ordered by increasing size, starting with one element sets. Then, the iterations computing subsets run until no new subsets appears (lines 3-5 in Algorithm 1). At each iteration, we run two iterators S_1 and S_2 over the sets under consideration. For every pair of S_1, S_2 we check if $S_2 \subset S_1$ and $|S_2| = |S_1| - 1$, and keep the cheapest S_{best} of all such subsets (lines 10-16). Additionally, the Banker's enumeration of sets [9] is performed. The Banker's enumeration iterates over subsets of the set of all predicates in graph R. Its main purpose is to generate the subsets of characteristic sets that are not characteristic sets themselves.

Figure 2: Hierarchical Characterisation

(a) HC



(b) Triple patterns and their optimal ordering



Algorithm 1: Compute Hierarchical Characterisation

Input: RDF graph R
Result: H – Hierarchical Characterisation of R

1 **begin**
2 $Sets \leftarrow$ `computeCharSets()`
3 **while** *Sets* $\neq \emptyset$ **do**
4 $H = H \cup Sets$
5 $Sets \leftarrow$ `computeSubsets(H)`
6 **return** H

7 COMPUTESUBSETS(SETS)
8 SubsetIterator si $\leftarrow$ init Banker's iteration
9 *res* $\leftarrow \emptyset$
10 **foreach** $S_1 \in Sets$ **do**
▷ iterate in increasing size
11 $S_{best} \leftarrow \emptyset$
12 **foreach** $S_2 \in Sets : |S_2| < |S_1|$ **do**
13 **if** $|S_2| = |S_1| - 1 \wedge S_2 \subset S_1$ **then**
14 update S_{best} to S_2 if its cost is the smallest
15 **while** *si.next()* $\neq S_2$ **do**
▷ get all subsets of S_1 that are missing from Sets
16 update S_{best} to $si.next()$ if its cost is the smallest
17 store the pointer to S_{best} in S_1
18 **if** $S_{best} \notin Sets$ **then**
19 res.insert(S_{best})
20 **return** res

Consider the following fragment of computation, where sets are listed in order of increasing size; sets printed in bold are characteristic, italic sets are the "missing" sets generated by the Banker's iterator si. Suppose that the current position of si is set $\{1, 2, 3\}$. Then, iterating until si reaches S_2 (line 15 of Algorithm 1), we encounter the set $\{1, 3, 5\}$ which is not characteristic (i.e., does not belong to the input `Sets`), but is a subset of S_1. If it is cheaper than the previously considered S_2, we update the S_{best} variable (line 16).

1 2 3
....
1 3 5 si
1 3 6 S_2
1 3 7
....
1 3 5 6 S_1
....

Although the Banker's iteration potentially enumerates all the subsets of all predicates in the dataset, in reality it stops relatively early, since it is always bounded by the largest set in `Sets` (see condition in line 14). This largest set gets smaller with every iteration, since every iteration considers subsets of sets generated in the previous iteration. Additionally, the biggest portion of the HC is identified during the first iteration, and the process converges really quickly. We also note that the set inclusion check in the innermost **for** loop (lines 12-16) is implemented extremely efficiently using Bloom filters. In our experiments, computing the Hierarchical Characterisation of the Uniprot dataset (with over 850 million tripes stored on disk) is done within 6 iterations and takes ca.700 ms.

4.3 Join Ordering for Star Queries

We first focus on finding the optimal join order in (sub)queries of the form

select *
where $\{?s\ p_1\ ?o_1.\ \ldots\ ?s\ p_k\ ?o_k\}$

Let $S = \{p_1, \ldots, p_k\}$ be the corresponding set of predicates. Our main idea is to extract the part of the Hierarchical Characterisation of the dataset starting with the set S. Namely, we find the set $S_1 \in H_0$ such that $S_1 = S$, get its cheapest subset S_2 (remember that S_1 has a pointer to S_2) and find out the predicate p in $S_1 \setminus S_2$. This predicate p corresponds to one of the triple patterns of the query ($?s$, p, $?o$); we put this triple pattern to be the *last* in the join order. The procedure repeats with S_2: follow the pointer to its cheapest subset S_3, put the triple pattern with the predicate from $S_2 \setminus S_3$ to be the *last but one* predicate in the join order. The process terminates when the current set S_k has only two elements: these predicates correspond to triple patterns that will be joined first. The pseudo-code of the algorithm is depicted in lines 1-9 of Algorithm 2. We note that if there is no set in H_0 that contains all the predicates from P (i.e., the lookup in line 3 fails), then the corresponding query yields an empty result.

The intuition behind this approach is the following: starting from the set of triple patterns, we find out what is the most expensive triple pattern, and schedule it to be the last in the join order. This expensive triple pattern is exactly the one that does not appear in the cheapest subset (w.r.t. our cost function) of the predicate set. Following this logic at each step, we construct the join tree top-down. An illustration of this algorithm is given in Figure 2b. The set with ID 154 (see Figure 2a) has the predicates from all four given triple patterns. Its cheapest subset in the HC is the set 27. Therefore, the triple pattern with the predicate *type* will be the last one in the join order (the upper-most triple pattern in the join tree). Similarly, the cheapest subset of the set 27 is the set with ID 6, and the missing predicate in it is *bornIn*. Since the last set has two predicate, they form the first join in the join ordering (lower-most

Algorithm 2: Join Ordering for Star Queries

```
Input: SPARQL star-shaped graph Q^R
Result: Ordering of joins
1  begin
2      P ← {p_1, ..., p_k}                          ▷ set of predicates in Q^R
3      S ← getHierarchicalCharSet(P)
4      O ←empty list                                 ▷ join ordering
5      while S.size() > 2 do
6          Subset ← S.Subset                         ▷ cheapest subset
7          p ← S \ Subset                            ▷ next predicate in ordering
8          O.push_front(p)
9          S ← Subset
       ▷ the two elements left in S = {p_1, p_2}
10     O.push_front(p_1); O.push_front(p_2)
11     if Q^R has constants then
12         if one of the constants is a key then
13             push it to the first level of the tree
14             return;
15         else
16             push down constants until
               cost(IndexScan) > cost(⋈)
```

level in the tree).

We note that this strategy yields the optimal join ordering for star-shaped queries in linear time. Besides, it does not assume independence between predicates in different triple patterns (unlike Dynamic Programming and the bottom-up greedy heuristics). It is therefore best suited for dealing with structural correlations that are so common in RDF data.

Unfortunately, this no longer holds if the query has constant objects, i.e. when some of $?o_1, \dots, ?o_k$ are replaced with literals or URIs. We have to, therefore, rely on a heuristics. It seems impossible to precompute all the correlations between constant objects and predicates in all sets of Hierarchical Characterisation. However, in real datasets we observed that some predicates in the sets of HC are extremely selective (like keys in relational world), and then all other predicates nearly functionally follow the selective ones. In our example with books and people, the name of the author is nearly the key (same with the title of a book). This can be captured while constructing the HC, if we track the multiplicity of each predicate in the sets. Then, the 'key' predicates are those with the multiplicity of 1. Note that we only need to store this multiplicity information for the sets from H_0, i.e. the characteristic sets.

Now, to construct the join ordering for triples with bounded objects, we first order the joins as if all objects were unbound. Then, we distinguish between two cases:

1. one of the bounded objects is in the triple with the 'key' predicate (lines 12-14). The entire star query is therefore a lookup of properties of a specific entity. We push down this triple pattern (basically by appending it at the front of our join ordering), and stop.
2. otherwise, we keep pushing down the constants in the join tree and stop when the cost of the corresponding index scan is bigger than the cost of the join on that level of the tree (lines 15-16).

We do not want to simply push down all constants, since some of the object constants (especially for the *type* predicate) are quite unselective and it is possible that the lowest join in the tree produces less tuples than the index scan on such unselective constant alone.

So far we have considered star queries centered around subject. Such patterns are extremely common in RDF datasets and are prevalent among the queries. However, the same techniques work for stars around objects (i.e., based on the object-object join). Since these queries are still valid SPARQL, the system derives and stores Hierarchical Characterisations for object-centered stars, too.

5. QUERY OPTIMIZATION OF ARBITRARY QUERIES

In this section we describe the algorithm for join ordering in general SPARQL queries. Our main idea is to decompose the query into star-shaped subqueries connected by chains, and to collapse the subqueries into meta-nodes. The star-shaped subqueries are optimized by the algorithm from Section 4. Then, the Dynamic Programming algorithm is run on top of the simplified query. In order to enable accurate cardinality estimations in the simplified query, we introduce a novel synopsis (*Characteristic Pairs*) that captures structural correlations in the RDF graph beyond star subqueries.

5.1 Characteristic Pairs

While some of the correlations between triples are captured by Characteristic Sets (define types in the RDF dataset) and consequently Hierarchical Characterisation (inheritance between different types, subject to a specific cost function), we are still missing other relationships between different types.

Let us illustrate it with an example. Consider the triples describing the person and its birthplace:

(s_1, hasName, "Marie Curie"), (s_1, bornIn, s_2),
(s_2, label, "Warsaw"), (s_2, locatedIn, "Poland")

There, the object id s_2 in the triples describing the person is used to link it to the city. In a way, this correspond to the "foreign key" concept in relational databases, except that of course RDF does not require to declare any schema. Mining these foreign keys thus becomes a challenge for the system. Knowledge of such dependencies is, on the other hand, extremely useful for the query optimizer: without it, the optimizer has to assume independence between two entities linked via *bornIn* predicate, thus almost inevitably underestimating the selectivity of the join of corresponding triple patterns. In reality, almost every person has information about the place of birth, so the selectivity of the join is close to 1.

In order to capture these "foreign key"-like relationships between nodes in the RDF graph, we will store pairs of characteristic sets that occur together (i.e., are connected by an edge) in the RDF graph, along with the number of occurrences. More formally, for the subject s let $S_c(s)$ denote the characteristic set of edges emitting from s (in other words, $S_c(s)$ is a type of s). We define a Characteristic Pair as

$$P_C(S_c(s), S_c(o)) = \{(S_c(s), S_c(o), p) \mid S_c(o) \neq \emptyset \wedge \exists p : (s, p, o) \in R\}$$

The condition $S_c(o) \neq \emptyset$ includes only those objects that appear as subjects in other triples, and therefore have non-empty characteristic sets. The set of all characteristic pairs is then defined as

$$P_C(R) = \{P_C(S_c(s), S_c(o)) \mid \ S_c(o) \neq \emptyset \wedge \exists(s, p, o) \in R\}$$

Additionally, we define the number of occurrence of a characteristic pair $P \in P_C(R)$ to be $count(P) = |\{(s, o) | P_C(S_c(s), S_c(o)) = P\}|$. Using this aggregate we can distinguish between "one-to-one" and "one-to-many" relationships. Namely, the proportion

$$\frac{count(P_C(S_c(s), S_c(o)))}{count(S_c(s))}$$

tells us, to how many objects of the same type does the subject s connect.

Although in theory, with n distinct characteristic sets we can get up to n^2 characteristic pairs, in real datasets only few pairs appear frequently enough to be stored. For YAGO-Facts dataset, out of ca.250000 existing pairs, only 5292 pairs appear more than 100 times in the dataset. This way, the frequent characteristic pairs for the YAGO-Facts consume less than 16 KB. We do not store the pairs with $count$ being smaller than 100. For such pairs, the independence assumption provides a "close enough" estimate of the result size. The optimizer will, most likely, underestimate the size to be just 1 tuple, but we found that misestimation in 100 tuples does not lead to a plan whose performance would differ from the optimal one.

5.2 Estimating Cardinalities using Characteristic Pairs

We start with considering the simplest example of a query that joins two star-shaped subqueries:

select distinct ?s ?o
where { $?s\ p_1\ ?x_1.$
$?s\ p_2\ ?x_2.$
$?s\ p_3\ ?o.$
$?o\ p_4\ ?y_1.$ **}**

In order to estimate the result size of the query, we simply need to find all the characteristic pairs consisting of sets that contain the predicates p_1, p_2, p_3 and p_4:

$$\text{cardinality} = \sum_{\{\{p_1,p_2,p_3\}\subset C_1, \{p_4\}\subset C_2\}} count(P_C(C_1, C_2)),$$

where C_1 and C_2 are the characteristic sets that contain p_1, p_2, p_3 and p_4, respectively.

Note that this computation does not assume independence between any predicates, and works for stars of arbitrary size. However, this simple estimation is only possible due to the `distinct` keyword: in general, the query can produce duplicate results for $?s$ and $?o$ for two reasons: first, there may be multiple bindings for $?x_1$, $?x_2$ and $?y_1$; second, there may exist multiple bindings of $?o$ for the same $?s$.

In order to cope with the first issue (multiple bindings for objects), predicates in characteristic sets are annotated with their number of occurrences in entities belonging to this set [11]. Similarly, we annotate the predicate that connects two entities in the characteristic pairs with its number of occurrences. Formally, for $P = (S_c(s), S_c(o), p)$ we compute $count(p) = |\{(s,p,o) \mid (s,p,o) \in R \wedge P_C(S_c(s), S_c(o)) = P\}|$. In other words, based on $count(p)$ we can derive whether the "foreign key" relationship between s and o is a "one-to-one" or "one-to-many". Namely, if $count(p) = count(P_C(S_c(s), S_c(o)))$, it is one-to-one, and with $count(p) > count(P_C(S_c(s), S_c(o)))$ it is one-to-many.

Consider again the query above but without the `distinct` modifier, and suppose that characteristic sets for s and o, and the characteristic pair are as depicted in Figure 3:

distinct	p_1	p_2	p_3
1000	1000	3000	2000

(a) CS for $?s$

distinct	p_4
5000	5100

(b) CS for $?s$

distinct	p_3
1000	2000

(c) $P_C(?s, ?o)$

Figure 3: Representation of two char.sets and a char.pair

The first column of each table gives us the number of `distinct` stars and pairs of stars This means that, on average, one entity of type $S_c(s)$ has $\frac{1000}{1000} = 1$ predicate p_1, $\frac{3000}{1000} = 3$ predicates p_2 and $\frac{2000}{1000} = 2$ predicates p_3, and on average for every s there are $\frac{2000}{1000}$ occurances of o connected to it via p_3 (see Table in Figure 3c). We can therefore estimate the cardinality of the query without the `distinct` as:

$$\underbrace{1000}_{\text{distinct}} \cdot \underbrace{\frac{1000}{1000} \cdot \frac{3000}{1000} \cdot \frac{2000}{1000}}_{?s} \cdot \underbrace{\frac{5100}{5000}}_{?o} \cdot \underbrace{\frac{2000}{1000}}_{\text{count}(p_3)} = 12240$$

This computation has to be corrected in case some of x_i or y_i are bounded. Specifically, for every constant x_i (y_i, respectively), we multiply our estimate by the selectivity of an object given the fixed predicate $sel(?o = x_i | ?p = p_i)$, i.e. the probability of the object restriction given the adjacent predicate restriction.

5.3 Join Ordering in Arbitrary Queries

Our join ordering strategy for general SPARQL queries is given in Algorithm 3. The algorithm starts with clustering the query into disjoint star-shaped subqueries (lines 11-24). In order to do it, we order the triple patterns in the query by subject (line 13), and group triple patterns with identical subjects (line 15). These groups potentially form star-shaped subqueries. Then, for every group of triple patterns we estimate its cardinality using characteristic sets. If its small enough (in our experiments we used a cutoff of 100K tuples), the group is turned into star subquery and the corresponding edges are removed from the query graph (lines 17-19).

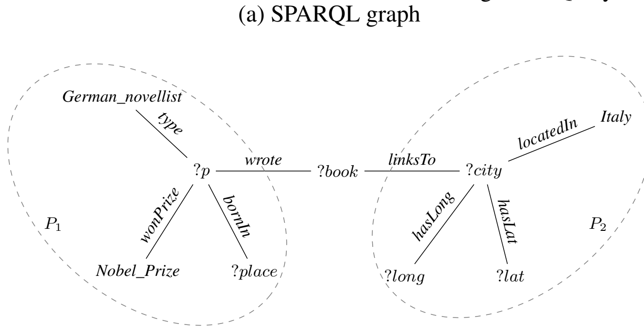
Consider the query from Figure 1a as an illustration. Its triple patterns after grouping look as follows:

```
star1 { ?p type German_novellist.
        ?p hasWonPrize Nobel_prize.
        ?p bornIn ?place.
        ?p created ?book.
      ?book linksTo ?city.
star2 { ?city isLocatedIn Italy.
        ?city hasLongitude ?long.
        ?city hasLatitude ?lat.
```

The corresponding subgraphs of the SPARQL query graph are denoted as P_1 and P_2 in FIgure 4a. Note that we don't form a star query around the variable $?book$, although syntactically it is a star with one edge. The reason is, based on our characteristic set estimation, we see that this star would return a lot of intermediate results (in fact, all the triples with the *linksTo* predicate). The algorithm stays conservative and does not "oversimplify" the query, leaving more choices to the later Dynamic Programming stage.

Figure 4: Query Graph Decomposition

(a) SPARQL graph



(b) Simplified SPARQL graph

$P_1 \xrightarrow{wrote} ?book \xrightarrow{linksTo} P_2$

(c) Entries of the DP table (predicate names denote the corresponding triple patterns)

Entities	Partial Plan	Cost
$\{P_1\}$	$(wonPrize \bowtie type) \bowtie bornIn$	3000
$\{P_2\}$	$(locatedIn \bowtie hasLong) \bowtie hasLat$	5000
$\{book\}$	IndexScan($P = linksTo$, $S =?book$)	0
$\{P_1, book\}$	$((wonPrize \bowtie type) \bowtie bornIn) \bowtie wrote$	3500
...	...	...

After the stars around subjects have been formed, we attempt to form star around objects on remaining edges. We give preference to subject-centered stars since in our experience they are more frequent in queries, and it is often more beneficial to execute them before object-centered stars.

The algorithm then considers all the stars formed by the `GetStars` subroutine; for every star it adds the new meta-node to the query graph and removes the intra-star edges (lines 4-5). The plan for the star subquery is computed using the Hierarchical Characterisation (see Algorithm 2) and added to the DP table along with the meta-node (lines 6-7).

After all the star subqueries have been optimized, we add the edges between meta-nodes to the query graph, if the original graph had edges between the corresponding star sub-queries (line 8). The selectivities associated with these edges are computed using the Characteristic Pairs synopsis, and the regular Dynamic Programming algorithm starts working on this simplified graph (line 10).

An example of the simplified query graph is given in Figure 4b. There, two meta-nodes P_1 and P_2 are connected with the chain. Although the $?book$ variable did not form the star subquery, we can still use its emitting predicate $linksTo$ to estimate the cardinality of the remaining joins in the query. For instance, in order to estimate the size of the join of P_1 and *book*, we look up all Characteristic Pairs such that the first set in the pair has predicates from P_1 (*type*, *wonPrize*, *bornIn* and *wrote*), and the second contains the *linksTo* predicate.

Finally, the first four entries of the DP table are depicted in Figure 4c. The optimal plan for the *book* entity is an index scan, at the same time for P_1 and P_2 we have plans for the corresponding star queries. Since the complexity of DP grows exponentially, even a small reduction of the query graph can greatly improve the performance of the overall join ordering strategy. In our case, simplifying the query graph from 8 nodes to 3 nodes gives a reduction from 8!=40320 plans to 3!=6 plans.

Algorithm 3: General SPARQL join ordering algorithm

Input: SPARQL graph Q^R
Result: Join ordering for Q^R
1 **begin**
2 $stars \leftarrow$ `getStars(`Q^R`)`
3 **foreach** $s \in stars$ **do**
4 add meta-node for s to Q^R
5 remove joins of s from Q^R
6 $p \leftarrow$ `GetStarJoinOrder(s)`
7 DPTable.push_back($s, p, cost(p)$)
8 add edges between adjacent meta-nodes
9 estimate selectivities using Char.Pairs
10 run DP algorithm on DPTable
11 GETSTARS(Q^R)
12 $stars \leftarrow \emptyset$
13 order triple patterns from Q^R by subject
14 **foreach** *distinct subject* $s \in Q^R$ **do**
15 $star \leftarrow$ triples with subject s ▷ using CharSets
16 $card \leftarrow$ `getCardinality`($star$)
17 **if** $card \leq budget$ **then**
18 $stars$.add($star$)
19 remove edges within $star$ from Q^R
20 order triple patterns Q^R by object
21 **foreach** *distinct object* $o \in Q^R$ **do**
22 $star \leftarrow$ triples with object o
23 . . . ▷ similar to lines 14-16
24 **return** $stars$

6. RELATED WORK

While query optimization in general (and join ordering in particular) is an old and well-established field, the SPARQL-specific issues have not yet attracted a lot of attention.

The RDF-3X [13] employs the exact Dynamic Programming algorithm, which faces computational problems when the query size grows. Its variant, DP with Characteristic Sets, uses more accurate selectivity estimations at the expense of even slower query compile time. The Jena optimizer [17] relies on the greedy join ordering heuristical algorithm, but it tends to underestimate intermediate result sizes [11], which can degrade the query execution time by orders of magnitude. A variant of the greedy algorithm that operates on the data flow graph of the query and takes into account different SPARQL constructs (like OPTIONAL and UNION) is proposed in [5].

Our work differs from the original Characteristic Set approach [11] in three main points: first, we extend the CS to the hierarchy and propose a linear-time join ordering heuristics based on the Hierarchical Characterisation; second, we introduce a novel statistical structure to capture the "foreign-key"-like relationships in RDF graph; finally, we suggest the query simplification algorithm for general SPARQL queries. Apart from the cardinality estimation problem, the Characteristic Sets can be used as a foundation for the physical layout of the RDF store [14].

The recently proposed Heuristical SPARQL Planner [18] intro-

duces several heuristical rules of ordering the joins based merely on the structure of the SPARQL query. However, completely ignoring the data statistics leads to unpredictably bad query plans.

Another approach towards SPARQL optimization is to translate the SPARQL query to its SQL equivalent, and then optimize this intermediate SQL query [6]. This, however, prevents the optimizer from using RDF-specific information about correlations between predicates in star subqueries and chains. Besides, straighforward translation of all SPARQL joins to SQL joins leaves the relational optimizer with the search space of enormous size.

Our query simplification techniques are similar to ones in [10]. They also reduce the search space size by making some simplification before the DP algorithm starts. However, our simplification is driven by different priciples: namely, we are using the SPARQL-specific star-subqueries as building blocks for the DP algorithm.

7. EXPERIMENTS

In this section we evaluate the quality of our algorithm, both in terms of its own runtime and the runtime of the produced query plans. We use three large real-world RDF datasets, and generate a large number of random queries (star-queries and queries of arbitrary shape) against them.

7.1 Algorithms and the runtime system

To study the performance of our algorithm, we implemented it in the RDF-3X system [13], and compared it with original RDF-3X optimizer (which in turn we studied in two variants: with and without Characteristic Sets [11], denoted DP and DP-CS). In addition, we implemented in RDF-3X the greedy heuristics of [7] (Greedy), and the heuristical SPARQL planner (denoted HSP) [18]. All join ordering algorithms have access to full stack of RDF-3X indexes (including all six permutations of *S*, *P* and *O*, and aggregated indexes). All produced plans are run with the same runtime settings (including the Sideways Information Passing [12]), so the difference between different running times is solely due to the quality of different optimizers. For all the plans, we disable the dictionary ID mapping, since its runtime depends only on the result size (and is the same across all algorithms).

The RDF-3X system is run on a server with two quad-core Intel Xeon CPUs (2.93GHz) and with 64GB of main memory using Redhat Enterprise Linux 5.4.

7.2 Datasets and Workload generator

We used three large real-world datasets from different domains: the knowledge base Yago[8] with over 110 million triples, the book-cataloging social network dataset LibraryThing [13] with 36 million triples and the biological dataset Uniprot [2] consisting of around 850 million triples.

In order to test the scalability and robustness of the query optimization algorithms, we generate the query workload for these three datasets. The generated queries are of two kinds, the star-shaped queries, and the arbitrary (complex) queries.

To generate the star-shaped queries, we first extracted a set of "central" nodes. In graph theory, the centrality of the node is typically defined based on its distance to all other nodes. Since our graphs are simply too large to compute centrality exactly according to any of the definitions from literature, we rely on a very simple heuristical node selection. Namely, we collect all the pairs of nodes (ls, ro) as a result of the join operation $(ls, lp, lo) \bowtie_{lo=rs}$ (rs, rp, ro) over the entire triple store. Since we are only interested in ls and ro, the join is actually performed on the aggregated indexes and is quite efficient. These pairs (ls, ro) define all the two-hop chains in the graph. We then select all the nodes n that appear as ls in one pair and as ro in some other pair. Intuitively, these are the nodes that participate in long chains, and are somewhere in the middle of these chains. The intuition behind such selection is that the nodes participating in long chains allow us to form complex queries around them, by combining star-shaped subqueries with chains connecting them.

As a result of the node selection procedure we are left with the handful of "central" nodes (from few hundreds in Yago to tens of thousands in LibraryThing). We then form the star-shaped queries around a subset of these nodes by randomly choosing the attributes of these nodes (i.e., the objects connected to it) and replacing some of them with variables. At the end we also replace the central node with the variable itself, so the query gets a form of a natural question "find all the entities with given attribute values, and extract some of their attributes".

In order to form arbitrary queries from the star-shaped queries, we expand some of variables (that replaced attributes of a central node) with either a chain, or another star. For example, if $?s$ is a central node in the star query and $?x$ is one of its attributes (so that there is a triple pattern $?sp?x$ in the query), we either attach a chain starting with $?x$, or another star centered around $?x$. The actual decision between a star and a chain is made at random; it also depends on whether $?x$ can actually form a chain or a star.

At the end, we group the star and arbitrary queries based on their size. The star queries have sizes from 5 to 10 triples (we filter out the smaller queries, since they are typically trivial to optimize), and the arbitrary queries contain from 10 to 50 triple patterns. For every group we collect 100 distinct queries.

Clearly, such strategy could choose some very unselective queries (e.g., find the names and addresses of all the people in the US). In order to prevent these queries in the workload, we run the candidate queries against the RDF-3X store and rule out those that take more than 30 seconds to finish. The plans for this candidate selection run are compiled using the DP and the greedy heuristics for large queries.

7.3 Methodology

For every query and every optimization strategy we run the optimizer 11 times, and record an average of all runs except the first one (this way we bring the system to the warm cache state). We call this number a *planning time* of the query; we excluded from it the (common for all optimizers) parsing, string-to-ID mapping and code generation time for SPARQL queries.

In theory, the Dynamic Programming strategy is supposed to yield the optimal plan, so we could just record how much slower the other techniques are (including ours, which is also a heuristics for arbitrary queries). But in reality the DP strategy often fails to find the best solution, because of cardinality misestimations during the cost function computation. Besides, sometimes the plan with the best cost function value does not yield the best runtime (although we do record the Pearson correlation between the runtime and the cost function (1) of 88%, so the cost function is not entirely misleading).

Therefore, to measure the quality of output plans for different query optimizers, we define the $rank$ of the algorithm A_i for the given query q as a proportion of the runtime of its output $plan(A_i, q)$ to the best runtime for that query among all algorithms A_i:

$$rank(A_i, q) = \frac{runtime(plan(A_i, q))}{\min_i runtime(plan(A_i, q))},$$

where $runtime(plan(A_i, q))$ is in turn the minimum among 10 warm cache runs of the $plan(A_i, q)$ in the system. For the group of queries (e.g., all star queries of size 8), we report the geometric

Table 1: Total Execution Time for Star Queries, Average over 300 Random Queries per Dataset, ms

	Query Size (number of joins)								
	Yago			*Uniprot*			*LibraryThing*		
	Total Execution Time (out of this: Optimization Time)								
Algo	$[5, 6]$	$[7, 8]$	$[9, 10]$	$[5, 6]$	$[7, 8]$	$[9, 10]$	$[5, 6]$	$[7, 8]$	$[9, 10]$
DP	92 (2)	129 (2)	241 (45)	94 (2)	142 (8)	257 (45)	95 (2)	140 (8)	266 (44)
DP-CS	103 (17)	170 (57)	393 (215)	110 (18)	186 (58)	421 (220)	105 (16)	188 (61)	404 (208)
Greedy	98 (1)	142 (4)	186 (8)	94 (1)	151 (4)	239 (9)	95 (1)	140 (4)	230 (8)
HSP	90 (0.2)	130 (0.2)	345 (0.3)	95 (0.2)	157 (0.2)	703 (0.3)	93 (0.2)	134 (0.2)	499 (0.3)
Our	**87** (1)	**120** (2)	**184** (4)	**93** (1)	**132** (3)	**207** (4)	**91** (1)	**135** (3)	**222** (4)

Table 2: Ranking of Runtime of Star Queries, Average over 300 Random Queries per Dataset

	Query Size (number of joins)								
	Yago			*Uniprot*			*LibraryThing*		
Algo	$[5, 6]$	$[7, 8]$	$[9, 10]$	$[5, 6]$	$[7, 8]$	$[9, 10]$	$[5, 6]$	$[7, 8]$	$[9, 10]$
DP	1.2	1.2	1.31	1.19	1.21	1.24	1.18	1.17	1.20
DP-CS	**1.15**	**1.13**	**1.19**	**1.18**	**1.16**	**1.18**	**1.13**	**1.13**	**1.15**
Greedy	1.3	1.38	1.31	1.21	1.33	1.35	1.19	1.21	1.20
HSP	1.2	1.3	2.3	1.22	1.39	3.8	1.18	1.19	2.7
Our	**1.15**	1.18	1.20	**1.18**	1.17	1.19	1.15	1.17	1.18

mean of $rank(A_i, q)$ taken across all queries q_i in that group for each algorithm separately. This will indicate the 'average ranking' of the query optimization strategy for the given group of queries (ranging from 1 to infinity, lower values are better).

7.4 Computing statistics

First we measure the time and space needed to store our hierarchical characterisation – the data structure used for the star queries optimization (and subsequently for optimizing arbitrary queries). Table 5 presents the numbers observed while loading the three datasets in RDF-3X. As we see, the Hierarchical Characterisation has the smallest footprint in Uniprot dataset. The reason for this small size is probably the well-structured nature of Uniprot which basically has a regular schema. This is the ideal case from the indexing standpoint, and the HC works very well in such a setting. The other two datasets are less regular, with more different entities types, but still the overall impact on the database size and loading time is rather small.

Table 5: Hierarchical Characterisation: Indexing Space and Time

Dataset	Space		Time	
	Mb	% of Total Size	s	% of Total Loading Time
Yago	5	0.1%	91	3%
Uniprot	0.6	0.0001%	10	0.001%
LibraryThing	15	0.5%	62	4%

Similarly, the time and space needed to index and store all the frequent Characteristic Pairs is very modest comparing to the overall loading time and the database size. As Table 6 shows, the extreme case here is again represented by Uniprot, which is a very structured RDF dataset. Yago and Librarything are somehow less structured, but still the amount of additional information is less than 0.01% of the overall database size, and the indexing time is practically negligible.

7.5 Query Optimization: Star Queries

In this section we summarize our experiments with star query optimization. First of all, Table 1 reports the total running time (i.e., optimization and plan execution time) for 300 randomly generated star queries per dataset. The queries are grouped by size, from small (5-6 joins) to large (9-10 joins). The reported time is the sum of the join ordering time and plan execution time, with join ordering time also being reported separately in the brackets. We see that, in terms of the overall runtime, our algorithm outperforms even the exact DP and DP-CS strategies. When it comes to just the query optimization time, the clear winner is the HSP planner with all compile times being much less than 1 ms. However, it pays a high price for that, since most of the time the generated plans are far from optimal. The Greedy strategy and our query decomposition take comparable amount of time, while the DP and DP-CS are the slowest, as expected. Note that the significant difference between planning times of DP and DP-CS is due to usage of Characteristic Sets for cardinality estimations. Indeed, for any particular query the CS-based cardinality estimation is rather fast, but it has to be performed for *every* subquery of the query graph. This renders the technique unfeasible for large query graphs.

Table 6: Characteristic Pairs: Indexing Space and Time

Dataset	Space		Time	
	Mb	% of Total Size	seconds	% of Total Loading Time
Yago	0.6	0.01%	1.2	0.008%
Uniprot	0.01	0.00001%	0.7	0.004%
LibraryThing	0.18	0.0003%	1.5	0.03%

The rankings of runtimes for different algorithms are given in Table 2. Somewhat surprisingly, our algorithm actually outperforms the Dynamic Programming strategy even without considering the join ordering time. This should be attributed to misestimations of cardinalities (and therefore the cost function) that the DP strategy makes during the join order construction. We also found that while for small queries (5 and 6 triple patterns), especially if they have constants, the performance of HSP is comparable with the rest of the algorithms, the large queries can get unpredictably bad plans from the HSP planner. The reason lies in the fact that HSP ignores all the available statistics and makes 'blind' decisions that could lead to extremely suboptimal plans.

To conclude, our query simplification approach yields the plans that are very close to the best ones, and it does so in a very reasonable amount of time.

Table 3: Ranking of Runtime of General Queries

Algo	Query Size (number of joins)											
	Yago				*Uniprot*				*LibraryThing*			
	[10, 20)	[20, 30)	[30, 40)	[40, 50]	[10, 20)	[20, 30)	[30, 40)	[40, 50]	[10, 20)	[20, 30)	[30, 40)	[40, 50]
DP	1.81	-	-	-	1.43	-	-	-	1.93	-	-	-
DP-CS	1.60	-	-	-	1.31	-	-	-	1.65	-	-	-
Greedy	2.13	1.83	1.98	3.17	1.67	1.90	2.01	2.05	1.88	1.96	2.15	2.44
HSP	3.01	7.08	5.94	11.51	3.31	4.98	3.36	8.91	5.33	6.28	11.72	8.15
Our	**1.50**	**1.45**	**1.25**	**1.38**	**1.18**	**1.21**	**1.13**	**1.15**	**1.45**	**1.31**	**1.19**	**1.24**

Table 4: Total Execution Time of General Queries, Average over 400 Random Queries per Dataset, ms

Algo	Query Size (number of joins)											
	Yago				*Uniprot*				*LibraryThing*			
	[10, 20)	[20, 30)	[30, 40)	[40, 50]	[10, 20)	[20, 30)	[30, 40)	[40, 50]	[10, 20)	[20, 30)	[30, 40)	[40, 50]
	Total Execution Time (out of this: Optimization Time)											
DP	7745 (7130)	-	-	-	9823(9323)	-	-	-	8603(7809)	-	-	-
DP-CS	65.7s(65.5s)	-	-	-	82.6s(82.1s)	-	-	-	69.5s(68.8s)	-	-	-
Greedy	857 (133)	1236 (413)	2204 (838)	4145 (1194)	739 (155)	1220 (422)	2092 (927)	2840 (1180)	912 (142)	1615 (414)	2644 (918)	3885 (1201)
HSP	1025 (2)	3189 (3)	4102 (4)	10720 (5)	1160 (2)	2094 (3)	1952 (4)	7228(5)	2187 (2)	3852 (3)	9415 (4)	8970 (5)
Our	**660** (150)	**967** (315)	**1211** (348)	**2174** (890)	**566** (153)	**838** (330)	**1356** (701)	**1755** (820)	**742** (148)	**1105** (302)	**1656** (697)	**2177** (813)

7.6 Query Optimization: Arbitrary Queries

In Table 4 we present the total execution times for general queries over three datasets. As for star queries, we report the *average* execution time over *400 randomly generated* queries per dataset; the queries (and corresponding runtimes) are grouped by the query size: from small (10 to 20 joins) to large (40 to 50 joins). Again, the given time is a sum of both optimization and execution time; the optimization time is also given separately in brackets after the total time. The exact algorithms, DP and DP-CS did not scale for the queries with more than 20 triple patterns (DP-CS takes more than 60 s on average for the queries with 10 to 20 joins), so we excluded them from the comparison for the rest of the queries.

The total execution time of our algorithm is consistently the best among all the competitors: it outperforms the exact DP and DP-CS for small queries due to more efficient search space enumeration, and gives better approximate solution to the join ordering problem than any other heuristics (Greedy and HSP). We see that the proposed algorithm scales linearly in the size of the query. Note that here again the HSP is the fastest algorithm to get the query plan, although the quality of its output is frequently the worst among all competitors, so the total execution time for HSP planner is the largest among all the heuristics and quickly gets worse as the query grows.

In order to study the quality of output plans in more detail, we present the ranking of execution time for general queries in Table 3. The plans produced by our algorithm outperform the competitors almost for all queries. For smaller queries (less than 20 triple patterns) it approaches the quality of the DP-based algorithms, and frequently outperforms them, since our Characteristic Pairs data structure captures correlations that are not available to the DP-based algorithms. For larger queries, it is up to 11 times better than the HSP algorithm. Note that using Characteristic Pairs with DP or DP-CS is not really feasible, since it would increase the compile time of this algorithms even more: the Pairs would have to be used to estimate the cardinality for *every* subplan under consideration, as opposed to our strategy to use it only between certain star-shaped 'blocks' of the query.

7.7 Effect of individual techniques

Finally, we quantify the effect of two techniques presented in the paper when used individually. Namely, we consider a query optimizer that (i) uses only Hierarchical Characterisation (*Only HC*); (ii) uses only Characteristic Pairs (*Only CP*); and (iii) uses both of them (*HC + CP*). Table 7 reports the total execution time of 400 random queries of arbitrary shape over YAGO dataset for these three setups.

We see that using the Characteristic Pairs in isolation from the query simplification and Hierarchical Characterisation is infeasible: indeed, without simplification the CP structure has to be used to estimate the cardinality of all the partial subplans during the query optimization (and their number grows exponentially). Using Hierarchical Characterisation alone seems more promising, as the query optimization time goes down a bit. However, without Characteristic Pairs the optimizer sometimes misses the optimal ordering of the star-shaped subqueries, resulting in suboptimal query plans. Finally, the combination of both techniques yields the best performance of the resulting plans.

Table 7: Evaluation of Individual Techniques: Total Runtime of 400 Random Queries, ms

Algo	Query Size (number of joins) *Yago*			
	[10, 20)	[20, 30)	[30, 40)	[40, 50]
	Total Execution Time (Optimization Time)			
Only CP	95876 (95332)	-	-	-
Only HC	715 (120)	1102(300)	1503 (332)	2309 (875)
HC +CP	660(150)	967 (315)	1211 (348)	2174 (890)

8. CONCLUSIONS

We introduced a novel join ordering strategy for SPARQL queries and showed that it outperforms the state-of-the-art solution in terms of quality of produced plans and the planning time. More specifically, the experiments have shown that it is possible to keep the planning time very low while obtaining the query plans close to optimal.

Our join ordering algorithm in its present form does not cover the OPTIONAL clauses of SPARQL, which are equivalent to the left outer joins and can not be freely reordered with other joins. Similar issue arises when considering SPARQL's property paths, which can be viewed as non-equi joins.

In this paper we consider static datasets. Since our data structures are merely statistical synopses for the query optimizer, the support for dynamic RDF datasets should include background bulk updates. These bulk updates include locating the characteristic sets touched by the updates, changing their counts and potentially reiterating over their subsets. The detailed procedures for such bulk updates are subject of future work.

9. ACKNOWLEDGMENTS

This work has been supported by the EU FP7 project LDBC.

10. REFERENCES

[1] AllegroGraph RDFStore Benchmark Results. http://www.franz.com/agraph/allegrograph/agraph_benchmarks.lhtml.

[2] Uniprot RDF. http://dev.isb-sib.ch/projects/uniprot-rdf/.

[3] M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An Empirical Study of Real-World SPARQL Queries. *CoRR*, abs/1103.5043, 2011.

[4] C. Bizer and A. Schultz. BSBM SPARQL Benchmark Results. http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/results/.

[5] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an Efficient RDF Store over a Relational Database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 121–132, New York, NY, USA, 2013. ACM.

[6] O. Erling. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.

[7] L. Fegaras. A New Heuristic for Optimizing Large Queries. In *DEXA'98*, pages 726–735. Springer-Verlag, 1998.

[8] J. Hoffart, F. M, S. K. Berberich, G. Weikum, J. Hoffart, K. Berberich, G. Weikum, and F. M. Suchanek. YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia. *Commun. ACM*, page 2009.

[9] J. Loughry, J. van Hemert, and L. Schoofs. Efficiently enumerating the subsets of a set. http://www.applied-math.org/subset.pdf.

[10] T. Neumann. Query simplification: graceful degradation for join-order optimization. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 403–414, New York, NY, USA, 2009. ACM.

[11] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, pages 984–994, 2011.

[12] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 627–640, New York, NY, USA, 2009. ACM.

[13] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, Feb. 2010.

[14] M.-D. Pham. Self-organizing structured RDF in MonetDB. In *ICDE Workshops*, pages 310–313, 2013.

[15] S. S. Sahoo. Semantic Provenance: Modeling, Querying, and Application in Scientific Discovery. PhD Thesis. 2010.

[16] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *Proc. VLDB Endow.*, 1(2):1553–1563, Aug. 2008.

[17] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 595–604, New York, NY, USA, 2008. ACM.

[18] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. Boncz. Heuristics-based query optimisation for SPARQL. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 324–335, New York, NY, USA, 2012. ACM.

[19] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, Aug. 2008.

[20] H. Williams, O. Erling, P. Duc, and P. Boncz. 150 Billion Triple dataset hosted on the LOD2 Knowledge Store Cluster. EU Project Report. http://static.lod2.eu/Deliverables/D2.1.4.pdf.

APPENDIX

The query with 18 triple patterns for the Yago dataset. It returns the information about the German writer, novels he created, and further art works linked to them:

```
select * where {
?s a yago:wikicategory_German_people_of_Brazilian_descent.
?s a yago:wikicategory_Nobel_laureates_in_Literature.
?s a yago:wikicategory_Technical_University_Munich_alumni.
?s yago:diedIn ?place. ?place yago:isLocatedIn ?country.
?s yago:created ?piece. ?piece yago:linksTo ?movie.
?movie a yago:wikicategory_1970s_drama_films.
?director yago:directed ?movie. ?director yago:hasWonPrize ?prize.
?piece yago:linksTo ?city. ?city yago:isLocatedIn yago:Italy. ?piece
yago:linksTo ?opera. ?opera a yago:wikicategory_Operas.
?s yago:influences ?person2.
?person2 a yago:wikicategory_Animal_rights_advocates.
?s yago:created ?piece3. ?piece3 yago:linksTo yago:New_York_City.
}
```