Heuristics-based Query Optimisation for SPARQL

Petros Tsialiamanis
ICS-FORTH
Heraklion, Greece
tsialiam@ics.forth.gr

Lefteris Sidirourgos
CWI
Amsterdam, the Netherlands
lsidir@cwi.nl

Irini Fundulaki
ICS-FORTH
Heraklion, Greece
fundul@ics.forth.gr

Vassilis Christophides
ICS-FORTH
Heraklion, Greece
christop@ics.forth.gr

Peter Boncz
CWI
Amsterdam, the Netherlands
boncz@cwi.nl

ABSTRACT

Query optimization in RDF Stores is a challenging problem as SPARQL queries typically contain many more joins than equivalent relational plans, and hence lead to a large join order search space. In such cases, cost-based query optimization often is not possible. One practical reason for this is that statistics typically are missing in web scale setting such as the Linked Open Datasets (LOD). The more profound reason is that due to the absence of schematic structure in RDF, join-hit ratio estimation requires complicated forms of correlated join statistics; and currently there are no methods to identify the relevant correlations beforehand. For this reason, the use of good heuristics is essential in SPARQL query optimization, even in the case that are partially used with cost-based statistics (i.e., hybrid query optimization). In this paper we describe a set of useful heuristics for SPARQL query optimizers. We present these in the context of a new *Heuristic SPARQL Planner* (HSP) that is capable of exploiting the *syntactic* and the *structural* variations of the triple patterns in a SPARQL query in order to choose an execution plan without the need of any cost model. For this, we define the variable graph and we show a reduction of the SPARQL query optimization problem to the *maximum weight independent set* problem. We implemented our planner on top of the MonetDB open source column-store and evaluated its effectiveness against the state-of-the-art RDF-3X engine as well as comparing the plan quality with a relational (SQL) equivalent of the benchmarks.

1. INTRODUCTION

During the last decade we have witnessed a tremendous increase in the amount of semantic data available on the Web in almost every field of human activity. More and more corporate, governmental, or even user-generated datasets break the walls of *private* management within their production site, and become available for future analysis by potential data consumer applications or services. For example, knowledge bases with billions of RDF triples from Wikipedia, U.S. Census, CIA World Factbook, open government sites in the US and the UK, national museums like the British Museum as well as international institutions, news and entertainment sources are published nowadays on the so-called Web of Data, along with numerous vocabularies and conceptual schemas from e-science, aiming to facilitate annotation and interlinking of both scientific and scholarly data.

This emerging global space, which connects data across domains, aims to support a new generation of *decision support* and *business intelligence* applications for individual users and communities in diverse areas. A central issue in this context is the meaningful manipulation and usage of large volumes of semantic data. In particular, we are striving for effective and efficient storage and querying techniques for semantic data expressed in RDF, the lingua franca of the Linked Open Data (LOD) initiative and hence the default data model for the Web of Data.

However, the current state of the art in the available commercial RDF stores still shows problematic query performance, typically caused by bad query plans, especially in complex queries such as those found in analytical workloads. Comparing to relational database systems, current SPARQL query optimizers are less mature, yet typically are faced with queries that consist of significantly more joins. This is due to the RDF data model, which eliminates an explicit schema underlying the data, such that every single accessed column in a query leads to yet another self-join to the so-called triple table, typically used for storing RDF data. In the case of RDF, the cost-based approach to query optimization – successful in the relational field – often does not give good results.

In many use-cases, where SPARQL users are accessing LOD data sources, typically reachable over an URI and often freshly (re-)loaded, the database may not have available statistics (e.g., histograms) needed for cost-based optimization. Moreover, in RDF it is not immediately clear on what to create statistics, as the data is essentially a directed labelled graph, where the same predicates may be used between multiple sub-classes of subjects/objects (and where these sub-classes are not explicitly declared or recognisable), and in which predicates themselves may also re-appear as subjects and objects, mixing data and metadata in this one big graph.

Even if we consider the extreme case of purely tabular data stored as RDF, such that the underlying graph is of perfectly regular shape, the job of a SPARQL query optimizer compared to a relational one is significantly more complex, not only because of the larger amount of (self-) joins, but also because it is not trivial to estimate the join hit-ratios in the SPARQL case. Whereas correlated cost estimation is considered a rare problem in relational optimisation, only necessary for special cases, it is a basic requirement for a cost-based SPARQL optimizer. The problem of keeping correlated join hit-ratio statistics is very hard to solve in the general case, as there are almost infinite potentially relevant correlations, such that it is not clear which statistics a SPARQL query optimizer should keep and search during query optimization. As a result, RDF stores, even if they rely on cost-based statistics for certain kinds of predicates (such as selections, or certain well-known joins) will in many other cases have to rely on *heuristics* anyway.

A different approach to solve this problem is to devise *heuristic-based* query optimization techniques without the need of any knowledge of the stored dataset. To this end, we propose the first *heuristic-based SPARQL planner (HSP)* that is capable of exploiting the syntactic and structural variations of the *triple patterns* in a SPARQL query in order to choose a near to optimal execution plan without the need of any statistics. Based solely on the syntax of a SPARQL query, we can decide which parts to evaluate first in order to quickly reduce the intermediate results. Similarly, we can decide the join order and maximize the number of merge-joins by looking at a variation of a SPARQL join graph, which we define as the SPARQL *variable graph*.

The heuristic-based optimisation techniques introduced in this work can be applied in a centralised but also in a distributed and parallel setting such as the Cloud. The main contributions of our work are:

- We propose a set of heuristics for deciding which triple patterns of a SPARQL query are more selective, thus it is in the benefit of the planner to evaluate them first in order to reduce the memory footprint during query execution. These heuristics are generic and can be used separately or complementary to each other, and also in traditional cost-based optimisers to create a hybrid planner.

- We propose the first *heuristics-based SPARQL planner (HSP)* based on these well-argued heuristics that exploit the syntactic and structural clues found in SPARQL queries. In particular, *HSP* tries to produce plans that maximise the number of merge joins, reduce intermediate results by choosing triples patterns most likely to have high selectivity, and determines the evaluation order based on the structural characteristics.

- To achieve the maximum number of merge joins, we define a new structure called SPARQL *variable graph*, which is a variation of the SPARQL join graph. We then present an original reduction of the query planning problem to the problem finding the *maximum weight independent set*. In a variable graph, nodes are query variables that are part of more than one join, and edges denote joins between these variables. The qualifying independent sets are translated to blocks of merge joins, connected between them with other types of more costly joins (e.g., hash joins) supported by the underlying engine.

- We implemented the *HSP* planner on top of an *open source columnar DBMS*, the MonetDB system [20]. We focused on the efficient implementation of *HSP* logical plans to the underlying MonetDB query execution engine, i.e., the physical algebra of MonetDB. The main challenge stems from the decomposed model of rows in a columnar database. A main difference between our plans and the plans produced by the cost-based standard SQL optimiser of MonetDB is that we produce *bushy* rather than *left-deep* query plans to facilitate the idiosyncrasies of SPARQL query plans.

- We have experimentally evaluated *the quality* and *execution time* of the plans produced by *HSP* with the state-of-the-art cost-based dynamic programming algorithm (CDP) employed by RDF-3X [22] using synthetically generated and real RDF datasets. In all queries of our workload, HSP produces plans with the *same number* of *merge* and *hash* joins as CDP. Their differences lie on the selection of ordered variables, as well as the execution order of joins, which in turn affects the size of the intermediate results.

Compared to existing approaches for SPARQL query planning, HSP exhibits some original features: *a)* unlike most SQL-based SPARQL engines, such as SW-Store [3], Oracle RDF [7], Sesame [6], Virtuoso RDF [9], HSP is capable of rewriting SPARQL queries in order to exploit as much as possible the ordered triple relations, as well to impose selections and join ordering using RDF-specific heuristics, and avoids the false sense of precision of relying on purely relational cost-based methods (which fail to capture join-selection correlations prevalent in SPARQL queries); *b)* rather than relying on partial statistics on equi-selections, leaf-level joins and cached path expressions, as found in Hexastore [39], RDF-3X [22], and YARS2 [13], HSP shows how far one can get by relying *exclusively* on heuristics. Our experiments with available benchmarks, show that the query optimization results achieved by HSP are comparable with the state-of-the-art, and could only get better if combined with certain cost- and statistics-based approaches that apply to RDF, as used by the latter class of systems, to construct in the future hybrid optimization strategies.

The rest of the paper is organised as follows. Section 2 discusses related work and in Section 3 we shortly present the basics of RDF and SPARQL. In Section 4 we present the heuristics in which HSP is based on. Section 5 details the reduction to the maximum weight independent set problem for achieving the maximum number of merge joins and discusses our heuristic-based planner. In Section 6 we present our experimental findings and conclude in Section 7.

2. RELATED WORK

SPARQL query processing engines can be distinguished into two broad categories: *RDF native* and *SQL-based* ones. The former propose main-memory resident indexes for RDF triples which are employed during SPARQL processing (mostly for evaluating selections), whereas the latter store RDF data either in a large triple table (*spo*) or in smaller property tables (e.g., *so*) [34] and rely on the optimization techniques of the underlying DBMS to efficiently evaluate SPARQL queries. The majority of the systems replace constants (i.e., URIs and literals) appearing in RDF triples by identifiers using a mapping dictionary to avoid processing long strings.

YARS2 [13] is a native RDF processing system that builds in main memory a set of six sparse indexes on a subset of the combinations of RDF triple components. It also uses a keyword index to support efficient lookups of RDF constants. HPRD [5] instead uses only three triple indexes *spo, po, os* implemented as B+-trees as well as a path index to accelerate the evaluation of SPARQL path queries (i.e., queries that involve long chains of triple patterns). The matching data for each path query is extracted and stored in the path index to accelerate path evaluation. Consequently, the evaluation of path queries can be translated into the problem of subsequence matching. Finally, HPRD relies on information regarding the number of occurrences of triple patterns in an RDF dataset to estimate the size of the intermediate results and decide join ordering (similar to the aggregated indexes of RDF-3X [22]). Hexastore [39] is another native RDF processing system that builds six indexes for every possible collation order of triple components in addition to the indexes *so* of property tables. In contrast to these works, we are using six sorted relations stored as regular tables in MonetDB as access paths instead of indexes. In addition, we provide a heuristic-based algorithm for deciding how these access paths are exploited in query plans. Structured indices proposed for RDF graphs as GRIN [36] and BitMat [4] are outside the scope of our work.

RDF-3X [22, 23, 24] is a native-RDF system that relies heavily on the use of indexes to process SPARQL queries over compressed RDF triples. In particular, triples are compressed by lexicographically sorting them and storing only the changes between them. RDF-3X builds a *clustered* B+tree index with composite keys over every possible collation order of triple components. Furthermore, RDF-3X uses *aggregated indexes* for each of the three possible pairs of triple components and in each collation order (*sp*, *so*, *ps* etc.). Each index stores the two columns of a triple on which it is defined and an aggregated count that denotes the number of occurrences of the pair in the set of triples. Aggregated indexes that are organized in B+-trees, are much smaller than the full-triple indexes and are used to avoid decompressing duplicate triples in the final query results. In addition, RDF-3X builds all three one-value indexes that hold for every RDF constant the number of its occurrences in the dataset. Finally, it builds indexes on frequently occurring data paths that store exact join statistics for them. All the above indexes are exploited for query optimization. Despite the exhaustive indexing employed by RDF-3X, the size of the indexes does not exceed the size of the dataset thanks to the compression scheme. Query processing relies mostly on merge joins over the sorted indexes discussed previously. The query optimizer of RDF-3X (*CDP*) uses dynamic programming for the enumeration of plans. It relies on a cost model to estimate the number of intermediate results based on statistics. This information is used by the planner to decide the join order and algorithms to be used for join evaluation. In contrast, our heuristic-based SPARQL planner (*HSP*) produces plans with the same number of merge and hash joins using solely the heuristics described in Section 4. It is worth also noticing that unlike traditional SQL optimizers (featuring left-deep plans), both CDP and HSP produce bushy plans capable of executing the maximum number of identified merge joins. Finally, the work by Neumann et. al [22] extends RDF-3X by exploring sideways information passing run-time optimization techniques for scalable RDF query processing.

Hartig et. al. [14] discuss a SPARQL query graph model (SQGM) and a set of operators to model the SPARQL operations (join, union etc.). The work focuses mostly on how SPARQL queries are rewritten into SQGM ones. Similarly, Stocker et. al. [32] propose standard relational algebraic rewritings for SPARQL queries. Finally, Schmidt et al. [30] study the set of equivalences over the SPARQL algebra as well as well known to relational algebra rewriting rules. Moreover, this work proposes an approach to semantic query optimization, based on the classical chase algorithm that is orthogonal to the problem we are tackling in this work. None of the above works discusses how query plans (i.e., join orders and join variables) are found as we do in our work. In the work by Vidal et. al. [38] RDF triples are stored in a large triple table and a set of physical operators are proposed for efficiently implementing star-shaped queries. In this work, a randomized cost-based optimization strategy is adopted to determine the most cost-effective plan among a set of execution plans of any shape (bushy, left deep etc.). The cost-based optimizer uses statistics about the size of properties, and the selectivity of subjects and objects to determine the most prominent star-shaped joins. In our work, we are able to produce near to optimal plans without the use of any statistics, and we rely on the physical operators of MonetDB for evaluating the resulting query plans. Husain et. al. [15] discuss RDF query optimization in the cloud. The objective is to produce plans that mimimize the number of *jobs*. For this, the authors try to group together in a job as many joins as possible per join variable by employing the *early elimination heuristic*. We follow a similar approach in which we try to maximize the number of merge joins by grouping together the triple patterns that share a common variable. The ordering of joins for a specific job is chosen with the use of statistics whereas in our work the identification of join orders is done using only heuristics.

SQL-based SPARQL systems [2, 3, 7, 18] store RDF triples in large triple table [7, 18] or in property tables [2, 3]. Contrary to our work where we use a large triple table, SW-Store [2, 3] uses vertical partitioning to store RDF triples in the C-Store [33] column store database. Standard indexes on the *s* and *o* columns of property tables are implemented as *ordered* columns. SPARQL queries are translated into their equivalent SQL ones and query optimization is taken care by the C-store engine. In the work by Chong et. al. [7] RDF triples are stored in a giant triple table in Oracle DBMS [25]. Materialized join views on the triple table, and *so* materialized join views are built to speed up query processing. SPARQL queries are translated into SQL ones that employ the RDF_MATCH table function to evaluate the joins. This table function uses the materialized join views and Oracle's query optimization techniques for efficient query processing. Virtuoso [10] follows a similar approach to the previous one for the storage of RDF triples and the translation of SPARQL queries into their equivalent SQL ones. Lu et. al. [18] store RDF triples in a large triple table in DB2. SPARQL queries are translated into SQL ones in a form that allows them to be directly included as sub-queries of other SQL queries. Despite the elaborate cost-based query optimization techniques, commercial SQL optimizers are based on cost models that do not work well for RDF. This is due to the absence of a logical schema, which along with integrity constraints could be used to devise plans that would efficiently evaluate a very large number of self-joins. A standard relational optimizer can estimate only the cost of scan operators but does not have any information related to join patterns that appear in SPARQL queries. Stocker et. al. [32], Neumann et. al [23, 21] discuss cardinality estimation techniques for RDF data that could be used to enhance existing SQL optimizers for supporting efficient SPARQL processing. In our work we tackle this issue by proposing a number of *RDF-specific heuristics* rather than *RDF-specific statistics* embedded in relational optimizers which are expensive to build and maintain, especially for large scale and evolving RDF datasets.

3. RDF AND SPARQL

The Resource Description Framework (RDF) [19], a W3C recommendation, is used for representing information about Web resources. It enables the encoding, exchange, and reuse of structured data, while it provides the means for publishing both human-readable and machine-processable vocabularies. It is used in a variety of application areas, such as the Linked Data initiative [17]. Its aim is to connect different data sources on the Web, and it has become very popular by exposing many data sets using RDF. DBpedia, BBC music information [16], and government datasets are only few examples of the constantly increasing Linked Data Cloud.

RDF is based on a simple data model that makes it easy for applications to process Web data. In RDF everything we wish to describe is a *resource*. A resource may be a person or an institution, or the relation a person has with that institution. A resource is uniquely identified by its Universal Resource Identifier (URI). The building block of the RDF data model is a *triple*. A triple is of the form *(subject, predicate, object)* where the *predicate (p)* (also called *property*) denotes the relationship between *subject (s)* and *object (o)*. An RDF *graph* is a set of triples. The nodes of such a graph represent the subjects and objects, while the labeled edges the predicates. We give the following definition.

DEFINITION 1. *An RDF triple (subject, predicate, object) is any element of the set $\mathcal{T} = \mathbb{U} \times \mathbb{U} \times (\mathbb{U} \cup \mathbb{L})$, where $\mathbb{U}$ and $\mathbb{L}$ are disjoint, $\mathbb{U}$ is the set of URIs, and $\mathbb{L}$ the set of literals. A set of RDF triples is called an RDF* graph.

An example of a set of triples is shown in Table 1. These triples are part of the SP^{2}Bench SPARQL benchmark dataset [29].

SPARQL [27] is the official W3C recommendation for querying RDF graphs. SPARQL is based on the concept of matching *graph patterns*. The simplest ones are called *triple patterns*, and they resemble an RDF triple, but they may have a variable in any of the subject, predicate, or object positions. A query that contains a conjunction of triple patterns is called *basic graph pattern*. A basic graph pattern matches a subgraph of the RDF graph when variables of the graph pattern can be substituted with RDF constants (URI's and literals) in the graph. In order to define formally a triple pattern, in addition to the sets $\mathbb{U}$ and $\mathbb{L}$ we define an infinite set $\mathbb{V}$ of variables.

DEFINITION 2. *A SPARQL triple pattern is any element of the set $\mathcal{TP} = (\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{L} \cup \mathbb{V})$, where $\mathbb{V}$ is the set of variables.*

A SPARQL graph pattern is defined recursively as follows:

- A triple pattern P is the simplest graph pattern.
- If P_1 and P_2 are triples patterns, then expressions P_1 . P_2, P_1 OPTIONAL P_2, and P_1 UNION P_2 are graph patterns.
- If P is a graph pattern and C is a SPARQL built-in condition, then the expression P FILTER C is a graph pattern.

The SPARQL syntax follows the SQL select-from-where paradigm. The SELECT clause specifies the variables that should appear in the query results. Each variable in SPARQL is prefixed with character ?. The graph patterns of the query are defined with the WHERE clause. Finally, a FILTER expression specifies explicitly a condition on query variables. For example, the following SPARQL query asks for the year and the journal with title "Journal 1 (1940)" that was revised in "1942".

```
SELECT ?yr,?jrnl
WHERE {?jrnl rdf:type bench:Journal .
       ?jrnl dc:title "Journal 1 (1940)" .
       ?jrnl dcterms:issued ?yr .
       ?jrnl dcterms:revised ?rev .
       FILTER (?rev="1942") }
```

To simplify our study and presentation of our algorithms, we relax the notation and the definition of the SPARQL queries and restrict them to only *join queries*. Our simplification serves our purposes since the join and selection operations are paramount, due to their cost, to query optimisation.

DEFINITION 3. *A SPARQL* join query *is defined as a set of k triples patterns $\mathcal{Q} = \{tp_0, \ldots, tp_k\}$.*

Such a SPARQL join query has the simpler form:

SELECT $?u_1, ?u_2, \ldots$
WHERE $\{tp_1.tp_2.tp_3.\ldots\}$

where $?u_1, ?u_2, \ldots$ are variables, $tp_1, tp_2, \ldots$ are triple patterns as defined in Definition 3, and '.' is the join operator of SPARQL. In such a join query, a variable $?u$ that appears in many triple patterns $tp_1, tp_2, \ldots$ implies a join between these triple patterns. The set of variables that appear in the SELECT clause are called the *projection* variables and are part of the answer of the query.

The answer of a SPARQL query with a SELECT clause is a set of *mappings*, where a mapping (i.e., the SPARQL analog of the relational *valuation*) is a set of pairs of the form *(variable, value)*. For example, the result of the evaluation of the previous query example over the set of RDF triples in Table 1 is the following mapping:

{(?yr, "1940"), (?jrnl, sp2bench:Journal1/1940)}

4. OPTIMIZATION HEURISTICS

Due to the fine-grained nature of RDF data – where a triple is just a narrow tuple with three attributes – SPARQL queries involve a large number of joins. Such joins dominate the query execution time. In addition, RDF data does not come with schema or integrity constraints, therefore, a query optimiser cannot take advantage of such information to produce an efficient query plan. Another approach for query optimization is needed, one based on the observation that the syntactical form of a SPARQL query reveals information about the data to be accessed. We advocate the use of heuristics to determine the query execution plan, instead of maintaining costly statistics for the stored data. Due to the highly distributed, volatile, and ever-changing nature of semantic data, a cost-based optimizer is likely to under-perform more often because of outdated statistics.

A SPARQL join query consists of numerous costly joins. The first and foremost important goal is to maximise the number of merge joins in the query plan. A merge join in this context is most commonly a sort-merge join, or any other join that takes advantage of the existence of an index. In the next section we present our approach to achieve this goal. An equally important goal is to minimize intermediate results in order to minimize the memory footprint during query execution. This is achieved by choosing the most selective triple patterns to evaluate first. Traditionally, deciding which triple patterns are more selective relies on statistics. Here, we have compiled a set of heuristics, based on the syntactical form of triple patterns, to determine the more selective ones.

HEURISTIC 1 *(Triple pattern order).* Given the position and the number of variables in a triple pattern we derive the following order, starting from the most selective, i.e., the one that is likely to produce less intermediate results, to the least selective.

$$(s,p,o) \prec (s,?,o) \prec (?,p,o) \prec (s,p,?) \prec$$

$$\prec (?,?,o) \prec (s,?,?) \prec (?,p,?) \prec (?,?,?)$$

The above ordering is based on the observation that given a subject and an object there are only very few, if not only one, properties that can satisfy the triple pattern. Similarly, it is very rare that a combination of a subject and property has more than one object value. In the same line of thinking we derive the rest of the orders. There can only be few subjects that have the same value for a property, while there are more many subjects with the same property no matter the object value. Finally, if a query pattern has 2 variables, then objects are more selective than subjects, and subjects more selective than properties. An exception to this rule is when the property has the value rdf:type, since that is a very common property and thus these triples should not be considered as selective.

	subject (s)	predicate (p)	object (o)
t_1:	sp2b:Journal1/1940	rdf:type	sp2b:Journal
t_2:	sp2b:Inproceeding17	rdf:type	sp2b:Inproceedings
t_3:	sp2b:Proceeding1/1954	dcterms:issued	"1954"
t_4:	sp2b:Journal1/1952	dc:title	"Journal 1 (1952)"
t_5:	sp2b:Journal1/1941	rdf:type	sp2b:Journal
t_6:	sp2b:Article9	rdf:type	sp2b:Article
t_7:	sp2b:Inproceeding40	dc:terms	"1950"
t_8:	sp2b:Inproceeding40	rdf:type	sp2bInproceedings
t_9:	sp2b:Journal1/1941	dc:title	"Journal 1 (1941)"
t_{10}:	sp2b:Journal1/1942	rdf:type	sp2bJournal
t_{11}:	sp2b:Journal1/1940	dc:title	"Journal 1 (1940)"
t_{12}:	sp2b:Inproceeding40	foaf:homepage	"http://www.dielectrics.tld/..."
t_{13}:	sp2b:Journal1/1940	dcterms:issued	"1940"

Table 1: A set of RDF triples from the SP^{2}Bench Dataset

HEURISTIC 2 *(Distinct position of joins).* The different positions in which the same variable appears in a set of triple patterns captures the number of different joins this variable participates in. A variable that appears always in the same position in all triple patterns, for example as subject, entails many self joins with low selectivity. On the other hand, if it appears both as object and property, chances are the join result will be smaller. The following precedence relation captures this preference:

$$p \bowtie o \prec s \bowtie p \prec s \bowtie o \prec o \bowtie o \prec s \bowtie s \prec p \bowtie p$$

where s, p, o refer to the subject, property, and object position of the variable in the triple pattern. This ordering stems from our observations while studying RDF data graphs. RDF data graphs tend to be sparse with a small diameter, while there are *hub* nodes, usually subjects. As a result, query graph patterns that form linear paths are more selective.

HEURISTIC 3 *(Triples with most literals/URIs).* This heuristic is a special subcase of HEURISTIC 1 but can be used independently. Triple patterns that have the most number of literals and URIs – or symmetrically less variables – are more selective. This heuristic is similar to the bound as easier heuristic of relational query processing [37], according to which, the more bound components a triple pattern has, the more selective it will be.

HEURISTIC 4 *(Triples with literals in the object).* An object of a triple pattern may be a literal or a URI. In such case, a literal is more selective than a URI. This is true for RDF data because in many cases if a URI is used as an object, it is used by many triples.

HEURISTIC 5 *(Triple patterns with less projections).* This heuristic allows us to consider as late as possible the triple patterns that contain projection variables. In the case in which the compared sets of triple patterns have the same set of projection variables, we prefer the set with the maximum number of unused variables that are not projection variables.

The above heuristics can be used in combination or separately for determining the order in which triple patterns should be evaluated, and thus achieving smaller intermediate results. These heuristics are suitable for different planning approaches, such as distributed environment, or hybrid optimizers where a cost model and heuristics work together. In the next section we show how these heuristics can be employed in our heuristic SPARQL planner for choosing the best set of triple patterns to consider for merge joins.

5. HEURISTIC-BASED SPARQL PLANNER

Our main objective is to produce query plans with the *maximum number of merge joins*. Merge joins make use of the ordering of the joining attributes to achieve better execution times. In this work we assume that the RDF data are stored in a triple table, and that all possible ordering combinations are also present. This is a common tactic in state-of-the-art RDF storing solutions [9, 31]. We refer to these six orderings as *spo, sop, ops, osp, pos, pso*. Other systems use clustered B-trees [22], or vertical partitioning [2]. However, the design of our planner is such that it is easy to adjust to these different approaches for storing RDF data. The commonality is that they all provide various *access paths* to the stored data, that is they provide different ways to fast access data through indexes.

We reduce the problem of maximising the total number of merge joins to the problem of finding the *maximum weight independent sets* in an RDF *variable graph*. In graph theory, an *independent set* is a set of vertices, no two of which share an edge [12]. If each vertex of a graph G is assigned a positive integer (the weight of the vertex) the *maximum weight independent set problem* consists in finding independent sets of maximum total weight, which is an NP-hard problem in general [11] and remains NP-hard even under restrictions in the forms of graphs. However, the *variable graph* is much smaller, and with better structural properties, than an RDF join graph, and thus an independent set can be easily found in a few milliseconds in modern hardware.

Intuitively the reduction to the maximum weight independent set is equivalent to finding the largest groups of triple patterns that can be merge-joined on the same variable. The reduction is done by modelling the query as a *variable graph* where $i)$ nodes in the graph are the variables that appear in the triple patterns of a SPARQL graph, $ii)$ two nodes are connected if and only if they belong to the same triple pattern, and $iii)$ a node has a weight, which is the number of the triple patterns the corresponding variable appears in. Consequently, the nodes in the *variable graph* that are returned as part of an independent set, are the variables that are evaluated with merge joins.

More formally, a *variable graph* is defined as follows.

DEFINITION 4. *Let Q be a set of triple patterns of a SPARQL join query as defined in* DEFINITION 3. *The* variable graph $G(Q)$ *is a weighted graph $G(Q) = (V, E, \beta)$ where V is the set of nodes,*

```
Input: SPARQL join query $Q$
Output: mapping $\mathcal{M} : \mathcal{TP} \rightarrow (V, \mathcal{P})$
$\mathcal{C} \leftarrow \emptyset$, a set of candidate variables;
$T \leftarrow Q$, a set of triple patterns;
while $(T \neq \emptyset)$ do
    $\mathcal{I} \leftarrow \emptyset$, a set of sets of variables;
    $S \leftarrow \emptyset$, a set of variables;
    Let $G(T)$ be the variable graph constructed from the set of triple patterns in $T$;
    determine all Maximum Independent Sets;
    $\mathcal{I} \leftarrow$ MaxIndepentendSets$(G(T))$;
    if $(|\mathcal{I}| > 1)$ then
        $\mathcal{I} \leftarrow$ apply HEURISTIC 3 in $\mathcal{I}$;
        if $(|\mathcal{I}| > 1)$ then
            $\mathcal{I} \leftarrow$ apply HEURISTIC 4 in $\mathcal{I}$;
            if $(|\mathcal{I}| > 1)$ then
                $\mathcal{I} \leftarrow$ apply HEURISTIC 2 in $\mathcal{I}$;
                if $(|\mathcal{I}| > 1)$ then
                    $\mathcal{I} \leftarrow$ apply HEURISTIC 5 in $\mathcal{I}$;
                end
            end
        end
    end
    $S \leftarrow$ RandomChooseOne$(\mathcal{I})$;
    $C \leftarrow C \cup S$;
    remove triple patterns from $T$ that are covered by a variable in $S$;
    $T = T \setminus \{tp \mid T, vars(tp) \cap S \neq \emptyset\}$;
end
forall the $(c \in C)$ do
    Pick all triple patterns that have variable $c$
    $T \leftarrow \{tp \in \mathcal{Q} | c \in vars(tp), tp \notin \mathcal{M}.keys\}$;
    forall the $(tp \in T)$ do
        AssignOrderedRelation$(\mathcal{M}, tp, c)$ ;
    end
end
Assign to the remaining triple patterns an ordered relation;
forall the $(tp \in Q, tp \notin \mathcal{M}.keys)$ do
    AssignOrderedRelation$(\mathcal{M}, tp, nil)$;
end
```

Algorithm 1: HSP

$E \subseteq V \times V$ is the set of edges, and $\beta : V \rightarrow \mathbb{N}$ is a weight function.

The weight function β assigns to each node v of the variable graph a weight equal to the number of triple patterns that v appears in. The weight of the variable minus 1 captures the number of joins this variable participates in.

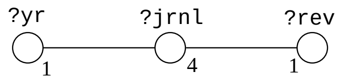


Figure 1: An example variable graph

Figure 1 shows the variable graph of the SPARQL join query example presented in section 3. There are 3 variables, namely `?jnrl, ?yr,` and `?rev`. Variable `?jnrl` is present in four triple patterns, hence it weight is 4, while the other two have a weight of 1. There are two edges connecting `?jnrl` with `?yr` and `?rev`, since they appear in triples together, but no edge between `?yr` and `?rev` since there is no triple containing both.

Notice that the *variable graph* is different than the RDF join graph. First, each variable in the variable graph appears only once, while in the join graph it will appear as many times as the joins it participates in. Second, the edges correspond to the triples and not to the join relationships. As a result, many joins of one variable collapse to only one node in the *variable graph*. For finding the maximum maximum weight independent sets of the *variable graph*, only the nodes that have weight greater than 2 will be considered, since only those are part of more than one join. For example, the *variable graph* of Figure 1 is trimmed down to only one node, namely `?jnrl`, since the weight of both `?yr` and `?rev` is 1. Consequently, the *variable graph* is much smaller than a join graph, and often much simpler to find the maximum weighted independent sets, despite the hardness of the problem. We demonstrate this also with our experimental evaluation in Section 6.

Next, we present the algorithm that implements the *Heuristic SPARQL Planner (HSP)* for deciding the merge joins and the ordered relations (i.e., access paths) that will be used to evaluate the query triple patterns.

Algorithm 1 depicts the *HSP* procedure in pseudo-code. HSP accepts as input a SPARQL join query Q, and returns a map $\mathcal{M}$, where each triple pattern of Q is mapped to an ordered relation and the variable that will be part of a merge join, or nil if there is no join. HSP first calls function `MaxIndepentendSets()` [26] to determine all maximum weighted independent sets of the variable graph $G(T)$ and stores them in $\mathcal{I}$. $\mathcal{I}$ is a collection of all maximum independent sets returned by function `MaxIndepentendSets()`. From all the possible candidates we have to choose one. Next, the HSP algorithm applies heuristics 3, 4, 2, and 5 in order to choose from all the different sets in $\mathcal{I}$ the one that is more selective, i.e., the one that produces the smallest intermediate results according to the heuristics presented in the previous section. If there are more than one sets left in $\mathcal{I}$ after the application of the heuristics, one set is picked randomly. Next, the triple patterns of Q that are used, i.e., are covered by a variable in the picked independent set, are removed from Q and the process is repeated for the remaining triple patterns in Q, until all triple patterns are covered.

The next step of the HSP algorithm determines which ordered relations should be accessed for each triple pattern according to HEURISTIC 1. For each variable that is part of the selected independent set, and for all triples that contain this variable, function `AssignOrderedRelation()` determines the correct ordered relation and updates the mapping structure $\mathcal{M}$.

Function `AssignOrderedRelation()` is shown in Algorithm 2. The input arguments are $i)$ the map structure $\mathcal{M}$ that stores the ordered relation and the variable v to be used for each triple pattern *tp* of Q, $ii)$ the triple pattern *tp*, and $iii)$ the variable v. If v is nil, then the triple pattern *tp* is not part of a join, but of a selection statement. Depending on the number of constants the appropriate order is chosen from the 6 available $(spo, pos, pso, osp, ops, sop)$. We use the help function $\texttt{pos}(tp, v)$ which returns either p, s, or o signifying the three possible positions, property, subject, or object, that variable v might occupy in the triple pattern tp. For example, assume the input triple pattern is of the form (l_1, u_1, l_2), where l_1, l_2 are constants and u_1 a variable, and v is nil. Then the ordered relation that should be accessed for this triple pattern is $(\texttt{pos}(tp, l_1), \texttt{pos}(tp, l_2), \texttt{pos}(tp, u_1)$. Now, $\texttt{pos}(tp, l_1) = s$ since l_1 appears in the subject position of the triple pattern. Similarly, $\texttt{pos}(tp, l_2) = o$, and $\texttt{pos}(tp, u_1) = p$. Hence, the ordered relation of this triple pattern is *sop*.

If v is not nil, then it participates in a merge join operation, thus the ordered relation is determined by picking the one that first orders the constants – if any – and immediately after the joining vari-

```
Input: map $\mathcal{M}$, triple pattern $tp$, variable $v$ of $tp$
if ($v = nil$) then
    if ($const(tp) = 2$) then
        $tp$ has 2 constants $l_1, l_2$ and 1 variable $u_1$;
        $\mathcal{M}.\mathtt{put}(tp \rightarrow$
        $((\mathtt{pos}(tp, l_1), \mathtt{pos}(tp, l_2), \mathtt{pos}(tp, u_1)), u_1))$;
    end
    if ($const(tp) = 1$) then
        $tp$ has 1 constant $l_1$ and 2 variable $u_1, u_2$;
        $\mathcal{M}.\mathtt{put}(tp \rightarrow$
        $((\mathtt{pos}(tp, l_1), \mathtt{pos}(tp, u_1), \mathtt{pos}(tp, u_2)), u_1))$;
    end
else
    if ($vars(tp) = 3$) then
        $tp$ has 3 variables $u_1, u_2, v$;
        $\mathcal{M}.\mathtt{put}(tp \rightarrow$
        $((\mathtt{pos}(tp, v), \mathtt{pos}(tp, u_1), \mathtt{pos}(tp, u_2)), v))$;
    end
    if ($vars(tp) = 2$) then
        $tp$ has 1 constant $l_1$ and 2 variables $u_1, v$;
        $\mathcal{M}.\mathtt{put}(tp \rightarrow$
        $((\mathtt{pos}(tp, l_1), \mathtt{pos}(tp, v), \mathtt{pos}(tp, u_1)), v))$;
    end
    if ($vars(tp) = 1$) then
        $tp$ has 2 constant $l_1, l_2$ and 2 variables $v$;
        $\mathcal{M}.\mathtt{put}(tp \rightarrow$
        $((\mathtt{pos}(tp, l_1), \mathtt{pos}(tp, u_1), \mathtt{pos}(v, tp)), v))$;
    end
end
```

Algorithm 2: AssignOrderedRelation

able, and last any remaining variables. For example, if the triple pattern is of the form (l_1, u_1, v) where l_1 is a constant and u_1, v variables, then since $\mathtt{pos}(tp, l_1) = s$, $\mathtt{pos}(tp, u_1) = p$, and $\mathtt{pos}(tp, v) = o$, the relation $(\mathtt{pos}(tp, l_1), \mathtt{pos}(tp, v), \mathtt{pos}(tp, u_1)) = sop$ is accessed.

After all triples are assigned an access path in the map structure $\mathcal{M}$, the join order has been determined and the HSP returns. Depending the underlying engine, a logical plan is produced.

6. EVALUATION

6.1 General Setup

To evaluate our work, we conducted the following two experiments: *(a)* we first compared the quality of the plans produced by our *heuristic-based* SPARQL Planner (HSP) with those produced by the *cost-based dynamic programming* planner (CDP) of RDF-3X [22] and *(b)* for each SPARQL query in our workload, we compared the time needed to execute in MonetDB the HSP plan translated into MonetDB's physical algebra (MAL), as well as an SQL translation of the SPARQL query, with the time needed by RDF-3X to evaluate the CDP plan. We choose to compare with RDF-3X because it is a state-of-the-art engine that relies heavily on statistics for query planning. However, we have not implemented HSP on top of RDF-3X because first, RDF-3X is a prototype implementation with no easy to separate software stack between the planner and the execution engine, and second, because RDF-3X relies so heavily in statistics that would call for a complete overhaul to remove those features and substitute them with only heuristics.

We rely on synthetic and real datasets and their query workloads for our experimentation. For this we used SP^{2}Bench [29] and YAGO [1].

All experiments were conducted on a Dell OptiPlex 755 desktop with CPU Intel Core 2 Quad Q6600 2.4GHz with 8MByte L2 cache, 8 GBytes of memory and running Ubuntu 11.04 2.6.38-8-generic x86_64. We used MonetDB5 11.2.0 [20] and RDF-3X version 0.3.5. MonetDB was extended with the Redland Raptor 1.9.0 [28] parser to parse the RDF triples and store them as regular tables in MonetDB. Both MonetDB and RDF-3X could import the datasets in less than half an hour and run the queries in the order of seconds. We performed only warm-cache experiments for which we ran the queries 21 times without dropping caches, we ignored the time of the first (cold) run and calculated the mean of the other 20 query runs.

6.2 Description of Datasets and Query Workload

In our experiments, we were able to scale SP^{2}Bench [29] synthetic data only up to 50M triples since RDF-3X was not able to load bigger datasets[1]. In order to load the YAGO dataset in MonetDB we had to manually perform some modifications: we removed a number of invalid characters contained in YAGO's URIs (e.g., $<$, $>$, etc.) that the Redland Raptor parser does not accept. In addition, RDF-3X ignores the base URI and consequently cannot distinguish between URI <`abc`> and literal "abc". We therefore converted all literals of the original YAGO dataset to URIs using as prefix the base URI of the corresponding RDF-XML file. By removing duplicate triples, we obtained a dataset containing 16M distinct triples. The modifications were necessary to ensure that both systems yield the same results for the same query.

Our study of datasets (for a complete report see [35]) confirmed the optimization heuristics we devised (Section 4). Regarding HEURISTIC 1 we observed that given a specific value for a subject and object, there are only few properties that satisfy the specific triple pattern. In addition, for given values for property and object in a triple pattern, we obtained a small number of subjects that satisfied it. Similarly, we found that it is very rare that a combination of a subject and property have more than one object value. An exception to this rule is when the property has the value `rdf:type`, since it is a very common property and thus these triples should not be considered as selective. The same was true for triple patterns that have specific values for their property and object components. Our findings were also verified by the study reported in [8]. In the case of HEURISTIC 2, we observed that join pattern $p \bowtie o$ returns always zero results making it the most selective one. The same is true for join patten $s \bowtie p$ for the SP^{2}Bench dataset, but not for the YAGO dataset (the only dataset where a URI can appear both as the property and subject of triples). Join patten $s \bowtie o$ returns always (significant) fewer results than the remaining join patterns (i.e., those that are specified on the same triple pattern component). More precisely, join $p \bowtie p$ yields results that are 1 to 2 orders of magnitude larger than $s \bowtie s$ and $o \bowtie o$ joins making it the least selective. Comparing the last ones, the former usually produces one order of magnitude less results than the latter.

To benchmark the plans produced by HSP and CDP, we have chosen to evaluate six conjunctive queries (and variations thereof) from SP^{2}Bench and four queries from YAGO benchmarks. Due to space limitation we do not present in detail the SPARQL syntax of all the queries we used. However, they can be found in [35] together

[1]According to RDF-3X installation instructions, RDF-3X cannot load datasets for which the data plus intermediate query results could exceed its virtual address space.

[2]Signifies the number of triple patterns that participate in the star join with the largest number of triples.

Query	SP1	SP2a	SP2b	SP3(a,b,c)_2	SP4a	SP4b	SP5	SP6	Y1	Y2	Y3	Y4
# Triple Patterns	3	10	8	2	6	5	1	1	8	6	6	5
# Variables	2	10	8	2	5	5	2	1	6	4	7	7
# Projection Variables	2	1	1	1	2	2	2	1	2	1	1	3
# Shared vars	1	1	1	1	5	4	0	0	4	3	3	4
# TPs with 0 const	0	0	0	0	0	0	0	0	0	0	2	3
# TPs with 1 const	1	9	7	1	4	3	1	0	6	3	2	0
# TPs with 2 const	2	1	1	1	2	2	0	1	2	3	2	2
# Joins	2	9	7	1	5	4	0	0	7	5	5	4
Maximum star join	2	9	7	1	1	2	0	0	4	3	2	1
Join Patterns												
# $s = s$	2	9	7	1	2	2	0	0	4	3	3	1
# $p = p$	0	0	0	0	0	0	0	0	0	0	0	0
# $o = o$	0	0	0	0	1	0	0	0	0	0	0	0
# $s = p$	0	0	0	0	0	0	0	0	0	0	0	0
# $s = o$	0	0	0	0	2	2	0	0	3	2	2	3
# $p = o$	0	0	0	0	0	0	0	0	0	0	0	0

Table 2: Query characteristics for SP^{2}Bench and YAGO

with all relevant details. As can be seen in Table 2 these queries involve a different number of triple patterns, variables and constants featuring selections as well as different kinds of joins among them (i.e., *star-* and *chain-shaped*) on different columnar positions (i.e. *s, p, o*). Variables which are not shared among triple patterns (i.e., join variables), or appear in SPARQL projections and filters are *unused*. We consider join queries that have different *structural characteristics* (i.e., kind of joins) and queries whose triple patterns have different *syntactic* characteristics (i.e, number of constants, shared variables and their positions in the pattern).

Clearly, queries SP5 and SP6 are the simplest ones, featuring selections with a different number of results. From the remaining ones involving joins, queries SP2a and SP2b of SP^{2}Bench contain a single large star query and their triple patterns are mostly syntactically similar (i.e., their constants are found in the same position) while YAGO queries Y1 and Y2 follow. Both contain medium star-joins with triple patterns that exhibit significant syntactical similarities. The remaining queries do not contain large stars, or in the case in which they do, the involved triple patterns exhibit syntactical dissimilarities.

In general our heuristics prove to be quite effective for queries whose triple patterns exhibit syntactical dissimilarities: they have different number of constants (and/or shared variables) that are found in different positions. In the following we discuss how HEURISTICS 1 to 4 are employed for each query in our workload.

We observed that the majority of the queries for both datasets considered $s \bowtie s$ joins (suggesting star-shaped joins on the *subject* component of the triple pattern), followed by $s \bowtie o$ joins. The smaller the ratio of shared variables over triple patterns, the heaviest are the star-shaped joins defined on the corresponding position of the triple pattern. This is the case of queries SP2a and SP2b, followed by query Y1.

Besides join algorithms and variables on which merge joins are performed (sorted variables), to compare the quality of the plans produced by HSP and CDP, we also estimated their cost using the cost model of RDF-3X [22]. In particular, we focus on the estimation of intermediate results of joins since the selection cost is asymptotically the same in both systems (logarithmic for binary search in MonetDB and for B+tree traversal in RDF-3X). Thus, in Table 3 we do not report the cost of simple selection queries SP5 and SP6. For the remaining join queries the costs of merge (depicted in bold face) and hash joins are estimated using the following CDP formula:

$$\begin{aligned} cost\_mergejoin(lc, lr) &= \tfrac{lc+rc}{100{,}000} \\ cost\_hashjoin(lc, rc) &= 300{,}000 + \tfrac{lc}{100} + \tfrac{rc}{10} \end{aligned}$$

where lc and rc are the cardinality of two join input relations, with the lc being the smallest one.

6.2.1 Query Plans

As can be seen in Table 4 for all the queries of our workload, our *heuristic-based SPARQL planner* (HSP) produces plans with the same number of *merge* and *hash* joins as the ones produced by the *cost-based dynamic programming planner* (CDP) of RDF-3X without the use of statistics. Their differences lie *only* on *join ordering* and the *type* of join that will be performed on each variable. These factors essentially affect the size of the intermediate results. The sorted variables on which merge joins will be performed are chosen early on by the maximum weight independent set algorithm. HEURISTICS 1 to 4 are then employed to determine the ordered relations on which the triple patterns will be evaluated as well as the join order. HSP heuristics are proved to be quite effective in choosing a *near to optimal plan* when queries exhibit *syntactical dissimilarities* (i.e. their triple patterns feature constants and variables in different positions).

More precisely, in the case of SP^{2}Bench queries SP1, SP3(a,b,c), SP4a, SP5, SP6 and YAGO query Y3, HSP produces exactly the same plans as CDP without using any cost-model. As a result the cost estimation of these plans is exactly the same in both systems (see Table 3). Furthermore, selections in HSP and CDP are evaluated for the same triple pattern on the *same* access path: ordered relation for HSP and full/aggregated index for CDP. For a subset of the queries and more specifically queries SP3(a,b,c) and SP6 of SP^{2}Bench, and Y3 of YAGO, CDP uses the aggregated index xy instead of the full triple index xyz. This is due to CDP's preference to use aggregated indexes when SPARQL triple patterns contain one or more unused variables in order to keep only the useful values. With the use of aggregated indexes CDP decompresses less triples for the scan and selection operations, obtains smaller intermediate results, and hence smaller input relations for the join operations.

Queries SP3(a,b,c) and SP4a are filtering queries. Unlike CDP, HSP systematically rewrites filtering queries into an equivalent form involving only triple patterns. CDP does not perform this rewriting. Instead, it executes an expensive join followed by the evaluation of the filter (queries SP3(a,b,c)). SP4a is a special case in which the

	SP1	SP2a	SP2b	SP3a	SP3b	SP3c	SP4a	SP4b
HSP	**32**	**873**	**830**	**487**	**100**	**105**	**354**+953,381	**264**+953,381
CDP	**32**	**31**	**54**	**487**	**100**	**105**	**354**+953,381	**299**+858,461

	Y1	Y2	Y3	Y4
HSP	**12**+300,054	**1**+303,579	**329**+302,577	**327**+763,749
CDP	**7**+300,023	**1.5**+301,614	**328**+302,577	**326**+763,603

Table 3: The cost of HSP and CDP plans

Query	SP1	SP2a	SP2b	SP3(a,b,c)_2	SP4a	SP4b	SP5	SP6	Y1	Y2	Y3	Y4
HSP												
Merge Joins	2	9	7	1	3	2	0	0	5	3	4	2
Hash Joins	0	0	0	0	2	2	0	0	2	2	1	2
Type of Plan	LD	LD	LD	LD	B	B	LD	LD	B	LD	B	B
CDP												
Mergejoin	2	9	7	1	3	2	0	0	5	3	4	2
Hashjoin	0	0	0	0	2	2	0	0	2	2	1	2
Type of Plan	LD	LD	LD	LD	B	B	LD	LD	B	B	B	B
Similar Plans	$\surd$	$\times$	$\times$	$\surd$	$\surd$	$\times$	$\surd$	$\surd$	$\times$	$\times$	$\surd$	$\times$

$LD: \; Left\ Deep\ Tree, \; B: \; Bushy\ Tree$

Table 4: Plan characteristics for SP^{2}Bench and YAGO

query (without the FILTER) contains a cross product. CDP recognizes the existence of the cross product at query compile time, and hence it does not produce any plan. To be able to benchmark CDP for these queries, we manually rewrote them into their equivalent form by eliminating the FILTER expressions. As reported in Table 4 HSP and CDP planners produce the same plan for queries SP3(a,b,c) comprising two selections and one merge join on the subject position of the triple patterns ($s \bowtie s$). On the other hand, for the light star query SP1 of SP^{2}bench that involves two $s \bowtie s$ joins, HEURISTICS 3 and 4 are used to determine join ordering, as well as the ordered relations on which selections are evaluated.

```
SELECT ?p
WHERE {?p ?ss ?c1 .                      (tp0)
       ?p ?dd ?c2 .                      (tp1)
       ?c1 rdf:type wordnet_village .    (tp2)
       ?c1 locatedIn ?X.                 (tp3)
       ?c2 rdf:type wordnet_site .       (tp4)
       ?c2 locatedIn ?Y.                 (tp5)
       }
```

Table 5: Yago Query Y3

SP^{2}Bench query SP4a and YAGO query Y3 are to a great extent syntactically dissimilar. SP4a contains small chain joins whereas Y3 contains small star joins (see Table 5) on variables found in different positions ($s \bowtie s$ and $s \bowtie o$). In addition, the triple patterns the join variables participate in, have different number of literals. Consequently, all our heuristics are effectively applied by HSP to produce the same bushy plan as CDP[3] (see Figure 2 for YAGO query Y3). In addition, both planners for SP4a and Y3 choose to execute the merge joins on the same variables. In the case of SP4a, and since HSP cannot estimate the number of intermediate results, it randomly selects one of the two possible choices for executing the hash joins that coincide for this query with the choices made by CDP.

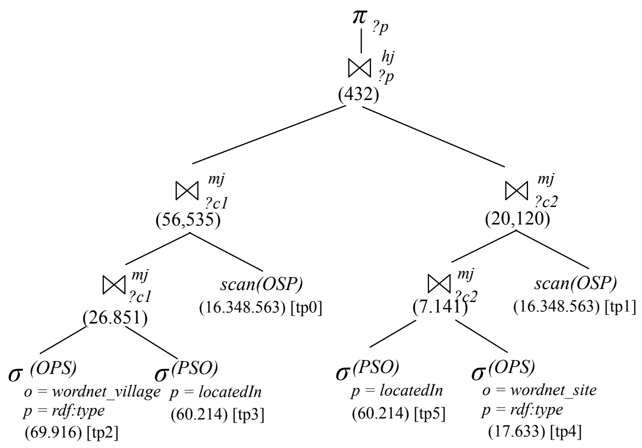


Figure 2: HSP Plan for YAGO query Y3

The queries for which HSP fails to decide a near to optimal plan are those that contain large star joins, with triple patterns that exhibit substantial syntactic similarities and consequently HSP heuristics are not very effective. This is the case of heavy star-shaped queries such as SP2a and SP2b.

For example, SP2a and SP2b queries form a join $s \bowtie s$ with very similar triple patterns (only HEURISTIC 3 is applied). HSP

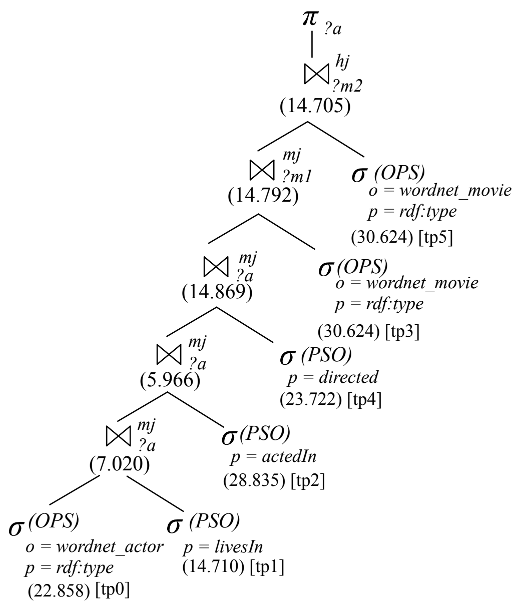
[3]In the query plans we write $\bowtie^{mj}_{var}$ and $\bowtie^{hj}_{var}$ to denote *merge* and *hash* joins respectively on variable var. We write $\sigma_{cond}(R)$ to denote a *selection* operation with condition $cond$ on a sorted relation (for HSP) or index (for CDP) R, and π_{vars} to denote a *projection* on the set of variables $vars$ of the input relation. For readability purposes we include below each operation the number of triples obtained by the evaluation thereof and when applicable the triple pattern concerned by the operation.

SP1	SP2a	SP2b	SP3a	SP3b	SP3c	SP4a	SP4b	SP5	SP6	Y1	Y2	Y3	Y4
0.10	0.15	0.13	0.09	0.09	0.09	0.13	0.12	0.06	0.06	0.13	0.12	0.14	0.13

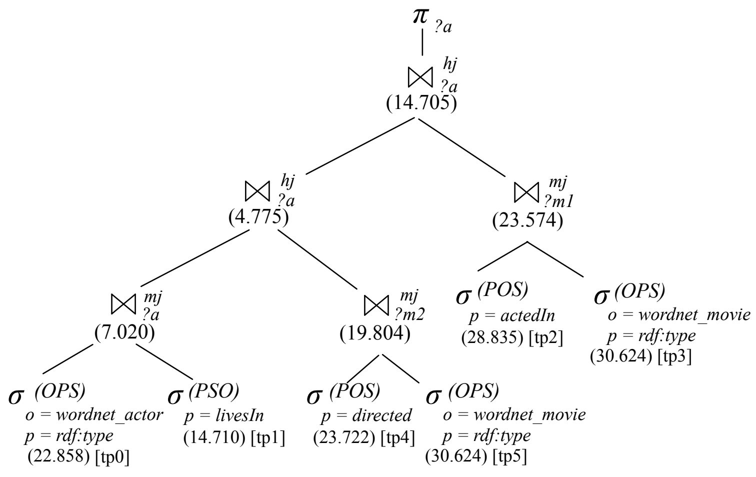
Table 6: Planning time of HSP for all queries (in ms).

	SP1	SP2a	SP2b	SP3a	SP3b	SP3c	SP4a	SP4b	SP5	SP6
MonetDB/HSP	19.52	3,267.01	1,035.12	80.92	8.74	12.55	3,602.09	1,766.29	0.06	0.43
RDF-3X/CDP	0.25	355.50	1,000.75	85.14	11.95	13.97	3,634.60	2,781.75	0.10	22.85
MonetDB/SQL	11.92	3,561	1,103	82.91	9.61	14.81	XXX	1,909.13	0.09	0.48

Table 7: Query Execution Time (in ms) for SP2Bench Queries (Warm Runs)


π ?a
hj ?m2
(14.705)
mj ?m1
(14.792)
σ (OPS) o = wordnet_movie p = rdf:type (30.624) [tp5]
mj ?a
(14.869)
σ (OPS) o = wordnet_movie p = rdf:type (30.624) [tp3]
mj ?a
(5.966)
σ (PSO) p = directed (23.722) [tp4]
mj ?a
(7.020)
σ (PSO) p = actedIn (28.835) [tp2]
σ (OPS) o = wordnet_actor p = rdf:type (22.858) [tp0]
σ (PSO) p = livesIn (14.710) [tp1]

(a) HSP


π ?a
hj ?a
(14.705)
hj ?a
(4.775)
mj ?m1
(23.574)
mj ?a
(7.020)
mj ?m2
(19.804)
σ (POS) p = actedIn (28.835) [tp2]
σ (OPS) o = wordnet_movie p = rdf:type (30.624) [tp3]
σ (OPS) o = wordnet_actor p = rdf:type (22.858) [tp0]
σ (PSO) p = livesIn (14.710) [tp1]
σ (POS) p = directed (23.722) [tp4]
σ (OPS) o = wordnet_movie p = rdf:type (30.624) [tp5]

(b) CDP

Figure 3: HSP and CDP plans for YAGO query Y2

```
SELECT ?a
WHERE {?a rdf:type wordnet_actor .        (tp0)
       ?a livesIn ?city .                 (tp1)
       ?a actedIn ?m1 .                   (tp2)
       ?m1 rdf:type wordnet_movie .       (tp3)
       ?a directed ?m2 .                  (tp4)
       ?m2 rdf:type wordnet_movie .       (tp5)
       }
```

Table 9: Yago Query Y2

correctly discovers the sorted variable on which the merge join will be performed, but chooses randomly among all possible join orders. The distinguishable characteristic for both queries is related to the size of the intermediate results that CDP uses to select the appropriate join ordering. SP4b is a complex star- and chain-shaped query for which HSP and CDP produce plans with the same number of merge and hash joins but defined on different variables. These planning decisions explain the differences in the estimated plan costs affecting more the evaluation of SP2a and SP2b than the evaluation of SP4b. As in the case of SP2a and SP2b, HEURISTIC 3 is the most effective.

For YAGO query Y1, HSP chooses to perform the majority of the involved merge joins on a single variable whereas CDP "breaks" this left deep subplan thus resulting in less intermediate results. In YAGO query Y2 (Table 9), HSP chooses to perform all the merge joins on one variable producing a left deep plan (see Figure 3(a)), whereas CDP produces a bushy one that reduces the size of intermediate results early in the plan (see Figure3(b)). In both queries, HSP heuristics are not very effective in discovering an interesting join order (except for HEURISTICS 3, 5 for Y1 and 3 for Y2) due to the syntactic similarities exhibited by the queries' triple patterns. Nevertheless, for the particular dataset the additional cost overhead is very small (see Table 3). Finally, YAGO query Y4 is a chain-shaped query consisting of 5 triple patterns, three of which do not contain any constant (making HEURISTICS 2, 3 the most effective ones). Hence, the query plan needs to scan the entire triple relation twice to evaluate the remaining patterns. Both HSP and CDP plans perform the merge joins on the same variables, and the only difference lies in the order of the two hash joins. As we can see in Table 3 the random choice of the order of hash joins does not seriously penalize the cost of the generated HSP plan.

We conclude this section by describing the SQL translation of SPARQL queries that will serve in the following as the *baseline* experiment for the plans produced by the standard MonetDB/SQL optimizer. Since unlike HSP and CDP, the MonetDB/SQL optimizer

	Y1	Y2	Y3	Y4
MonetDB/HSP	6.04	8.65	25.69	2.32
RDF-3X/CDP	15.75	9.95	81.20	90.45
MonetDB/SQL	7.69	9.07	538.65	1,113

Table 8: Query Execution Time (in ms) for YAGO queries (Warm Runs)

produces only left deep plans, we could not translate the SPARQL queries into SQL ones using exactly the same access paths as those employed by HSP. We simply choose to evaluate each triple pattern of the SPARQL query on the ordered relation that promotes the use of binary search for selections and returns the variable with the most number of appearances in the query sorted, to maximize (if possible) the number of merge joins. In the case in which a triple pattern contains constants, we chose the ordered relation according to HEURISTIC 1. Thus, the MonetDB/SQL optimizer will undertake the task of join ordering using runtime optimization techniques (e.g., sampling).

6.2.2 Query Execution Times

In this section, we report for each SPARQL query in our workload, the execution time of the HSP plan directly translated into MonetDB's physical algebra (MonetDB/HSP) and the execution time of the CDP plan evaluated by RDF-3X. We additionally report the execution time of its equivalent SQL rewriting when evaluated by the standard MonetDB/SQL optimizer as described previously.

We also present in Table 6 the planning time needed by Algorithm 1 alone, without evaluating the queries. The planning times for the HSP are very short (between 100 and 200 microseconds). Moreover, we expect the number of nodes on the variable graph to be kept always small, thus making the maximum independent set algorithm applicable. This is true because the *variable graph* consists only of the variables that appear twice or more in joins. Some more experimentation showed that HSP can process a variable graph of up to 50 nodes in less than 6ms. Such a graph implies at least 100 joins which is the common limit for other traditional optimizers found in relational engines.

The execution times for our experiment for the SP^{2}Bench and YAGO datasets are shown in Tables 7 and 8 respectively. The reported times do not include the planning time (less than 4% of the total execution time), the time to transform the constants of every triple pattern to ids as well as the conversion of these ids back to strings in the final query result. To speed up the resolution of the ids to URIs/literals and decompression thereof, RDF-3X sorts and groups the query results to decompress only one element per group of duplicates. This time is also not included in our measurements.

For the queries for which HSP and CDP produce the same plan (SP1, SP3(a,b,c), SP4a, SP5, SP6 and Y3), with the exception of SP1, MonetDB/HSP could be up to two orders of magnitude faster than RDF-3X/CDP (e.g., in SP6) and up to one order of magnitude faster than MonetDB/SQL (e.g., in Y3). In the case of SP1, although the HSP plan is the same as CDP, its execution in MonetDB is significantly slower. The same behaviour is also exhibited by MonetDB/SQL. This overhead cannot be justified by the *left-join* operators employed by MonetDB to get the subject values of a triple from the obtained object and property values and it is attributed to an internal bug. Note also that in query SP4a, the MonetDB/SQL optimizer chooses to execute a Cartesian product and thus fails to terminate. We can also observe that although in SP6 RDF-3X/CDP employs much smaller aggregated indexes, it is largely outperformed by MonetDB/HSP. This can be attributed to the large number of triples in the final result of SP6 (compared to the similar selection query SP5) for which decompression of the deltas of ids (for literals and URIs) needs to be performed. For the same reason performance gains are also exhibited in Y3 where MonedDB/HSP is 2.5 times faster than RDF-3X/CDP. This large difference in execution times for query Y3 is due to the fact that it contains two joins, where one of the two inputs is the entire relation. In addition, CDP uses in its plan aggregated indexes, and it takes a substantial amount of time to decompress them.

For queries with different plans, and especially in the case in which HSP selects a random order for executing the merge and hash joins, RDF-3X/CDP outperforms MonetDB/HSP up to one order of magnitude (e.g., in SP2a). When the triple patterns yield intermediate results of the same order of magnitude, join ordering has little influence on the query execution time (e.g., in SP2b). In the remaining queries although the HSP plans are not optimal w.r.t. CDP, query execution time in MonetDB/HSP is always better than RDF-3X/CDP up to one order of magnitude (e.g., in Y4). As a matter of fact, the execution of all query operators in MonetDB appears to be more efficient than in RDF-3X (exhibiting also the limitation in the size of the datasets that can be processed in main memory).

7. FUTURE WORK AND CONCLUSIONS

In this work we have introduced a set of heuristics for choosing which triple patterns of a SPARQL query are more selective by only looking at the syntactical form of the query, and not relying on any statistics of the stored data. We defined a variable graph based on the SPARQL join query and showed how to maximize the number of merge-joins by reducing this problem to the problem of maximum weight independent set. Based on these ideas, we introduced the first heuristics-based SPARQL optimizer, called HSP. We have performed an extensive evaluation on the quality of the plans generated by HSP. We also compared the query execution time achieved by HSP with two state of the art engines, namely RDF-3X and MonetDB.

We wish to investigate the effects of applying our heuristics in a distributed environment such as MapReduce and Hadoop. We also intend to extend our work to cope with different relational storage schemas, instead of only the traditional approach of a triple table. Also, we are working to integrate our solution with the MonetDB run-time optimizer in order to handle queries such as large star joins) for which our heurisics fail to produce near to optimal plans. Finally, we wish to extend our optimizer to include all features of the SPARQL language, such as the OPTIONAL clause.

8. REFERENCES

[1] Yet Another Great Ontology. http://www.mpi-inf.mpg.de/yago-naga/yago.

[2] ABADI, D. J., MARCUS, A., MADDEN, S. R., AND HOLLENBACH, K. Scalable semantic web data management using vertical partitioning. In *VLDB* (2007), pp. 411–422.

[3] ABADI, D. J., MARCUS, A., MADDEN, S. R., AND HOLLENBACH, K. SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management. *VLDB Journal 18*, 2 (April 2009).

[4] ATRE, M., CHAOJI, V., ZAKI, M. J., AND HENDLER, J. A. Matrix "Bit" loaded: A Scalable Lightweight Join Query Processor for RDF Data. In *WWW* (2010).

[5] BAOLIN, L., AND BO, H. HPRD: A High Performance RDF Database. In *Network and Parallel Computing* (2007), pp. 364–374.

[6] BROEKSTRA, J., KAMPMAN, A., AND VAN HARMELEN, F. Sesame: An Architecture for Storing and Querying RDF Data and Schema Information. In *Spinning the Semantic Web* (2003).

[7] CHONG, E. I., DAS, S., EADON, G., AND SRINIVASAN, J. An efficient SQL-based RDF querying scheme. In *VLDB* (2005).

[8] DUAN, S., KEMENTSIETSIDIS, A., SRINIVAS, K., AND UDREA, O. Apples and Oranges: A Comparison of RDF benchmarks and Real RDF Datasets. In *SIGMOD* (2011).

[9] ERLING, O., AND MIKHAILOV, I. RDF Support in the Virtuoso DBMS. In *In Proc. of the CSSW* (2007).

[10] ERLING, O., AND MIKHAILOV, I. RDF Support in Virtuoso DBMS. In *Networked Knowledge - Networked Media, SCI* (2009).

[11] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory qf NP-Completeness*. Freeman, 1979.

[12] GIBBONS, A. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[13] HARTH, A., UMBRICH, J., HOGAN, A., AND DECKER, S. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *ISWC* (2007).

[14] HARTIG, O., AND HEESE, R. The SPARQL Query Graph Model for Query Optimization. In *ESWC* (2007).

[15] HUSAIN, M. F., MCGLOTHIN, J., MASUD, M. M., KHAN, L. R., AND THURAISINGHAM, B. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *TKDE* (2011).

[16] KOBILAROV, G., SCOTT, T., RAIMOND, Y., OLIVER, S., SIZEMORE, C., SMETHURST, M., BIZER, C., AND LEE, R. Media Meets Semantic Web – How the BBC Uses DBpedia and Linked Data to Make Connections. In *ESWC* (2009).

[17] Linked data. http://linkeddata.org/.

[18] LU, J., CAO, F., MA, L., YU, Y., AND PAN, Y. An Effective SPARQL Support over Relational Databases. In *SWDB-ODBIS* (2007).

[19] MANOLA, F., MILLER, E., AND MCBRIDE, B. RDF Primer. www.w3.org/TR/rdf-primer, 2004.

[20] MonetDB. http://www.monetdb.org.

[21] NEUMANN, T., AND MOERKOTTE, G. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In *ICDE* (2011).

[22] NEUMANN, T., AND WEIKUM, G. RDF-3X: a RISC-style engine for RDF. *PVLDB 1*, 1 (2008).

[23] NEUMANN, T., AND WEIKUM, G. Scalable join processing on very large rdf graphs. In *SIGMOD* (June 2009), pp. 627–640.

[24] NEUMANN, T., AND WEIKUM, G. The RDF-3X engine for scalable management of RDF data. *VLDB Journal 19*, 1 (2010).

[25] Oracle database semantic technologies. http://www.oracle.com/technetwork/database/options/semantic-tech/index.html.

[26] OSTERGARD, P. R. J. A new algorithm for the maximum-weight clique problem. *Nordic Journal of Computing 8* (2001), 424–436.

[27] PRUD'HOMMEAUX, E., AND SEABORNE, A. SPARQL Query Language for RDF. www.w3.org/TR/rdf-sparql-query, 2008.

[28] http://librdf.org/raptor/.

[29] SCHMIDT, M., HORNUNG, T., LAUSEN, G., AND PINKEL, C. SP2bench: A SPARQL performance benchmark. In *ICDE* (2009).

[30] SCHMIDT, M., MEIER, M., AND LAUSEN, G. Foundations of SPARQL Query Optimization. In *ICDT* (2010).

[31] SIDIROURGOS, L., GONCALVES, R., KERSTEN, M., NES, N., AND MANEGOLD, S. Column-store support for RDF data management: not all swans are white. *PVLDB 1*, 2 (2008).

[32] STOCKER, M., SEABORNE, A., BERNSTEIN, A., KIEFER, C., AND REYNOLDS, D. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW* (2008).

[33] STONEBRAKER, M., ABADI, D., BATKIN, A., CHEN, X., CHERNIAK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S., E.O'NEIL, P., RASIN, A., TRAN, N., AND ZDONIK, S. C-Store: A Column Oriented DBMS. In *VLDB* (2005).

[34] THEOHARIS, Y., CHRISTOPHIDES, V., AND KARVOUNARAKIS, G. Benchmarking Database Representations of RDF/S Stores. In *ISWC* (2005).

[35] TSIALIAMANIS, P. Heuristic Optimization of SPARQL queries over Column-Store DBMS. Master's thesis, University of Crete, September 2011.

[36] UDREA, O., PUGLIESE, A., AND SUBRAHMANIAN, V. S. GRIN: A Graph Based RDF Index. In *AAAI* (2007).

[37] ULLMAN, J. D. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988.

[38] VIDAL, M.-E., RUCKHAUS, E., LAMPO, T., MARTINEZ, A., SIERRA, J., AND POLLERES, A. Efficiently Joining Group Patterns in SPARQL Queries. In *ESWC* (2010).

[39] WEISS, C., KARRAS, P., AND BERNSTEIN, A. Hexastore: sextuple indexing for semantic web data management. *PVLDB 1*, 1 (2008).