Collaborative Project

LOD2 – Creating Knowledge out of Interlinked Data

Project Number: 257943 Start Date of Project: 01/09/2010 Duration: 48 months

Deliverable 2.5

MonetDB Release with Optimized Graph Path Processing

Dissemination Level	Public
Due Date of Deliverable	01/09/2012
Actual Submission Date	01/09/2012
Work Package	WP 2, "Storing and Querying Very Large Knowledge Bases"
Task	T 2.3
Type	Report
Approval Status	Under review
Version	1.1
Number of Pages	50
Filename	LOD2_D25_MONETDB_RDF.doc

Abstract: This document describes the HSP heuristic SPARQL front-end for MonetDB that was used as a basis for experiments with RDF graph clustering; and an overview of this ongoing research in RDF graph clustering and indexing. It accompanies the recent availability of both the MonetDB RDF storage support and the HSP as open source packages.

Project funded by the European Commission within the Seventh Framework Programme (2007 – 2013)

History

Version	Date	Reason	Revised by
0.5	20/08/2012	Added first part (EDBT paper on HSP)	I. Fundulaki@cwi.nl
1.0	27/08/2012	Completed with RDF clustering & indexing approach	P.Boncz@cwi.nl
1.1	01/09/2012	Finished after comments by reviewer Vassilis Christophides	P.Boncz@cwi.nl

Author List

Organisation	Name	Contact Information
CWI	Peter Boncz	boncz@cwi.nl
CWI	Irini Fundulaki	I. Fundulaki@cwi.nl
CWI	Lefteris Sidirourgos	E.Sidirourgos@cwi.nl
CWI	Duc Minh Pham	duc@cwi.nl
*ICS-FORTH**	*Petros Tsialiamanis*	*tsialiam@gmail.com*
*ICS-FORTH**	*Vassilis Christophides*	*christop@ics.forth.gr*

** no direct authors of this deliverable, but included here for their co-authorship of the EDBT 2012 paper, which forms the basis of part 1 of this deliverable.*

Executive Summary

This deliverable marks a first description of support in RDF inside the open-source MonetDB with the MonetDB system. Its first part consists of material found in the 2012 EDBT paper "Heuristics-based Query Optimisation for SPARQL" that reports on a project where ICS-FORTH cooperated with CWI to create a SPARQL front-end for MonetDB with a heuristic approach to query optimization (the "HSP" front-end). This front-end was built on the RDF storage option that CWI added to MonetDB as part of its LOD2 activities in 2010/2011, and now have been put in the main MonetDB open source release.

It is now possible to perform SPARQL queries using MonetDB+HSP. The status of HSP is that of a research prototype, where only basic features of SPARQL1.0 are supported. Its main use has been as a platform for RDF query processing research. As such, it is not yet part of the LOD2 stack, where the components should be mature and functionally rich.

The HSP system has recently been released in open source under the GPL license by ICS-FORTH. For development purposes, the code is hosted at CWI in a Mercurial repository: ssh://hg@dev.monetdb.org/HSP

The RDF storage support of MonetDB has been committed into the trunk (main branch) of the common open-source MonetDB system which is hosted by CWI at: ssh://hg@dev.monetdb.org/MonetDB

As a result of the initial cooperation between FORTH and CWI, CWI invited dr. Irini Fundulaki for a 1-year sabbatical at CWI in the LOD2 project starting April 2012, under which the further open-sourcing activities of HSP have taken place. MonetDB+HSP also forms a platform for RDF/graph query processing, storage and indexing experiments at CWI. This further research will continue for the further duration of LOD2.

The second part of the deliverable document contains a description of this ongoing research. This research sketches a new perspective on the storage of RDF data, advocating the use of physical clustering techniques in order to improve the locality of SPARQL queries, specifically focusing on queries that touch large amounts of triples (i.e. analytical SPARQL queries).

For this research, initial experiments with MonetDB+HSP were performed. The promising outcomes of these tests will influence future SPARQL support in MonetDB, and this document contains an architectural outline of this future RDF support in MonetDB

Abbreviations and Acronyms

LOD	Linked Open Data
HSP	Heuristic SPARQL Planner
RDF	Resource Description Framework
YAGO	Yet Another Great Ontology – a large public RDF corpus
SP2Bench	A synthetic RDF benchmark based on the DBLP use case
CDP	Cost-based dynamic programming optimizer (such as used in RDF-3X)
RDF-3X	A leading research prototype for RDF storage and SPARQL querying
MonetDB	An open-source database system supported by CWI, employing columnar storage
DSM	Decomposition Storage Model, i.e. columnar storage
NSM	Normalized Storage Model: traditional row storage (record-based)
TPC-H	Standard relational decision support (=analytical) query benchmark.
BAT	Binary Association Table [pos,value] is the basic concept of a column object in MonetDB. It represents a value array consisting of a dense range of positions conceptually as a table.

Table of Contents

1. The HSP System

1.1 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. Knowledge bases with billions of RDF triples from Wikipedia, U.S. Census, CIA World Factbook, open government sites in the US and the UK, and international institutions, news and entertainment sources along with numerous vocabularies and conceptual schemas from e-science are published in the Linked Open Data Cloud.

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).

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, where an RDF triple (*subject*, *predicate*, *object*) indicates that *subject* is associated with *object* through *predicate*. This simple model eliminates an explicit schema underlying the data, such that every single accessed column in a query leads to yet another self-join on a very large *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 a URI and often freshly reloaded, 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 by resources that belong to belong to different classes (and where these classes are not explicitly declared or recognisable), and in which predicates themselves may also appear as both *subjects* and *objects*, mixing *data* and *schema* 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 case of SPARQL queries. 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.

Tsialiamanis et al. [40] discuss a different approach for solving this problem by devising a set of *heuristic-based* query optimization techniques without taking into advantage any knowledge of the stored dataset. To this end, they proposed 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, the authors can decide which parts to evaluate first in order to quickly reduce the intermediate results. Join ordering is also decided on the basis of the proposed heuristics by looking at a variation of the SPARQL join 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 the work discussed

in [40] are:

- A set of heuristics that exploit the *syntactic* and *structural* clues found in SPARQL queries for deciding which triple patterns of such a 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.
- HSP (Heuristic-based SPARQL planner) is the first heuristics-based SPARQL planner that builds on the proposed well-argued heuristics. 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. The query planning problem in HSP is reduced to the problem finding the maximum weight independent set in a graph. 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.
- The HSP planner was implemented on top of the MonetDB system [20], an open source columnar DBMS. The focus was 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 stemmed from the decomposed model of rows in a columnar database. A main difference between the produced HSP plans and the plans obtained by the cost-based standard SQL optimiser of MonetDB is that the HSP plans are bushy rather than left-deep plans. . Such types of plans facilitate the idiosyncrasies of SPARQL query plans namely joins that are translated into a large number of self-joins over a large triple table.
- The quality and execution plans produced by HSP were compared 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 the considered workloads, 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 proved to exhibit some original features:

- 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 that prevail in SPARQL queries);
- 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. The experiments conducted with available benchmarks, show that the query optimization results achieved by HSP are comparable with those presented in state-of-the-art works, 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 HSP system provides SPARQL1.0 functionality as a front-end to the MonetDB database system, which provides RDF storage for it (see Section 1.3.2). Its current status is a research prototype. The most striking limitations currently are: (i) lack of SPARQL1.1 features such as grouping and subqueries (ii) blank nodes are not supported in queries, and (iii) there is very limited support for literal types (all literals are treated as strings) and consequently also limitations the predicates and functions supported on literals.

1.2 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, HSP uses six sorted relations stored as regular tables in MonetDB as *access paths* instead of indexes. In addition, HSP [40] provides 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 the work by Tsialiamanis et. al.

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 employed 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, HSP produces plans with the same number of merge and hash joins using solely the devised heuristics. 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 in addition to well known 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 that is addressed by HSP. None of the above works discusses how query plans (i.e., join orders and join variables) are found as in

HSP. 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. On the other hand, HSP produces near to optimal plans without the use of any statistics, and relies 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 minimize 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. HSP follows a similar approach in which it tries 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 HSP 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 HSP that uses 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. HSP tackles 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.

1.3 Heuristic-based SPARQL Planner

1.3.1 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.

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 labelled edges the predicates.

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

	subject(s)	**property(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	sp2b:Inproceedings
t_9	sp2b:Journal1/1941	dc:title	"Journal 1 (1941)"
t_{10}	sp2b:Journal1/1942	rdf:type	sp2b:Journal
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 triples from the SP2Bench Dataset

An example of a set of triples is shown in Table 1. These triples are part of the SP2Bench 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 sub-graph 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 U and L we define an infinite set V of variables.

DEFINITION 2. A SPARQL triple pattern is any element of the set *TP* = (U ∪ V) × (U ∪ V) × (U ∪ L ∪ V), where V is the set of variables. SPARQL graph patterns are defined by means of the *join*, *optional* and *union* SPARQL operators.

Tsialiamanis et. al [40] discuss graph patterns defined using the *join* operator only. This simplification serves the purpose of the work since the join and selection operations are paramount, due to their cost, to query optimisation.

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 in the query's `WHERE` clause. Finally, a `FILTER` expression specifies explicitly a condition on query variables. For example, the following SPARQL Q1 query asks for the year and the journal with title "Journal 1 (1940)" that was revised in "1942".

Q1 : `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") }

Tsialiam et al. [40] represent a SPARQL join query as a set of triple patterns:

DEFINITION 3. A SPARQL join query is defined as a set of *k* triples patterns *Q = {tp0, . . . , tpk}.*

Such a SPARQL join query has the simpler form:

`SELECT` *?u1, ?u2, . . .*

`WHERE` {tp1.tp2.tp3. . . .}

where *?u1, ?u2, . . .* are variables, *tp1, tp2, . . .* are triple patterns as defined in DEFINITION 3, and '.' is the join operator of SPARQL. In such a query, variables that appear in more than one triple patterns *tp1, tp2, . . .* imply a join between these triple patterns. The projection variables are those that appear in the query's `SELECT` clause and are part of the answer of the query.

The answer of a SPARQL query of the above form 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*). Mappings are presented in tabular form, where the attributes are variable names. For example, the result of the evaluation of the previous query Q1 over the set of RDF triples in Table 1 is the mapping *{(?yr,* "1940"), *(?jrnl,* sp2bench:Journal1/1940)} shown also in tabular form:

?yr	?jrnl
"1940"	sp2bench:Journal1/1940

1.3.2 MonetDB RDF Storage

Since August 2012, the MonetDB open source trunk adds support for RDF storage, which was developed as a prelude to the work on HSP. This storage follows a rather classical approach that represents RDF as triples, which consist of three object identifiers (OIDs): S (Subject), P (Property) and O (Object). These OIDs are either all limited to 32-bits, or to 64-bits integers (we used always the 64-bits integers as these allow to store more than 2G triples). The main storage of RDF data is in 6 triple tables (SPO, SOP, PSO, POS, SPO and SOP) and hence in 18 columns (MonetDB is a column store).

The mapping from URI to OID is done by a special dictionary module, which decomposes URIs into parts, such that datasets where URIs have a large common prefix only store this prefix once. When a new URI is

parsed, it is added to the dictionary as the new suffix of the largest prefix of an already exiting URI that it has in common. The MonetDB RDF support for literals is currently limited to string literals only, which are handled by storing them in a mapping table that is kept in order. This allows range-restrictions on string literals to be translated into OID range restrictions. Literal OIDs are distinguished from URI OIDs as they have the highest bit set.

An important thing to note about the MonetDB RDF dictionary is that it assigns OIDs for URIs in an order that corresponds to parsing order (each literal and URI gets the OID of its first appearance). We will come back to this issue in Section 2.

1.3.3 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. Tsialiamanis et. al [40] 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. 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. Tsialiamanis et. al [40] have compiled a set of *heuristics* that are 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 the following order is proposed, starting from the most selective, i.e., the one that is likely to produce less intermediate results, to the least selective.

$$(s, p, o) < (s, ?, o) < (?, p, o) < (s, p, ?) < (?, ?, o) < (s, ?, ?) < (?, p, ?) < (?, ?, ?)$$

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. The rest of the orders were derived in the same line of thinking. 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 HEURISTIC 1 suggests that 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. A similar heuristic was proposed in Kaoudi et. al. [41].

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:

$$(p \bowtie o) < (s \bowtie p) < (s \bowtie o) < (o \bowtie o) < (s \bowtie s) < (p \bowtie p)$$

where *s*, *p*, *o* refer to the *subject*, *property*, and *object* position of the variable in the triple pattern. This ordering stems from the study of RDF data graphs. RDF data graphs tend to be sparse with a small diameter, while there are hub nodes, usually acting as *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 the triple pattern is.

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 considers 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, the set with the maximum number of unused variables that are not projection variables is preferred.

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.

1.3.4 Heuristic-based SPARQL Planner

The main objective of the HSP planner 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. Tsialiamanis et. al [40] store the RDF data in a triple table, and that all possible ordering combinations of *subject (s)*, *property (p)*, *object (o)* attributes are also present. This is a common tactic in state-of-the-art RDF storing solutions [9, 31]. These six orderings are referred to as *spo*, *sop*, *ops*, *osp*, *pos*, *pso*. Other systems use clustered B-trees [22], or vertical partitioning [2]. However, the design of HSP 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.

In the work of Tsialiamanis et. al [40], the problem of maximising the total number of merge joins is reduced 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. 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:

- nodes in the graph are the variables that appear in the triple patterns of a SPARQL graph
- two nodes are connected if and only if they belong to the same triple pattern, and
- 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. Even though finding maximum weight is an NP-hard problem, the variable graph that is used 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. The formal definition of a variable is 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, $E \subseteq V \times V$ is the set of edges, and $\beta : V \rightarrow N$ 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 shows the variable graph of the SPARQL join query Q1. 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.

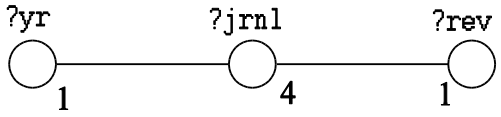

Figure 1: Variable Graph

The variable graph is different from 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 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.

The algorithm that implements the HSP planner decides the *merge joins* and the *ordered relations* (i.e., *access paths*) that will be used to evaluate the query triple patterns. The algorithm accepts as input a SPARQL join query *Q*, and returns a map *M*, where each triple pattern of *Q* is mapped to one of the ordered relations *(spo, sop, pos, pso, ops, osp)* and the variable that will be part of a merge join, or nil if there is no join. The algorithm computes the set of maximum weight independent sets [26] of the query variable graph. Since multiple such sets can be returned, the HEURISTICS proposed are applied in order to select one of the sets. Heuristics are applied in the following order: 3, 4, 2 and 5 to choose from all the different returned sets 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 I after the application of the heuristics, one set is picked randomly. The algorithm then uses the triple patterns of *Q* that are not associated with a variable in the picked independent set (the others are removed from the initial query *Q*) and the process is repeated for the remaining triple patterns in *Q*, until all triple patterns are considered.

The next step of the 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, the correct ordered relation is assigned to the variable and the mapping structure *M* that assigns to each variable an ordered relation is updated. In order to decide the ordered relation to be used for evaluating a triple pattern, the algorithm takes into account the number of constants/variables and their position in the triple pattern.

For example, assume the input triple pattern *tp* is of the form *(l1, u1, l2)*, where *l1, l2* are constants and *u1* a variable, and triple pattern *tp* does not participate in a join. Then the ordered relation that should be accessed for this triple pattern is *(pos(tp, l1), pos(tp, l2), pos(tp, u1)*, where *pos(tp, x)* returns the position of *'x'* in the triple pattern *tp*. Now, *pos(tp, l1) = s* since *l1* appears in the subject position of the triple pattern. Similarly, *pos(tp, l2) = o*, and pos(tp, u1) = *p*. Hence, the ordered relation of this triple pattern is *sop*.

If the triple pattern includes a join variable say *v* that participates in a merge join operation. Then the ordered relation is determined by picking the one that first orders the constants – if any –immediately after the joining variable, and last any remaining variables. For example, if the triple pattern is of the form *(l1, u1, v)* where *l1* is a constant and *u1, v* variables, then since *pos(tp, l1) = s*, *pos(tp, u1) = p*, and *pos(tp, v) = o*, the relation *(pos(tp, l1), pos(tp, v), pos(tp, u1)) = sop* is chosen.

After all triples are assigned an access path, the join order has been determined and HSP returns. Depending the underlying engine, a logical plan is produced.

1.4 Evaluation

Tsialiamanis et. al [40] conducted the following two experiments to evaluate the proposed approach : (a) they compared the quality of HSP plans with those produced by the cost-based dynamic programming planner (CDP) of RDF-3X [22] and (b) for each SPARQL query in the considered workload, 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 was compared with the time needed by RDF-3X to evaluate the CDP plan. The choice of RDF-3X is justified since it is a state-of-the-art engine that relies heavily on statistics for query planning. However, HSP was not implemented on top of RDF-3X because first, RDF-3X is a prototype implementation with no easy way 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.

Experiments were done using the synthetic SP2Bench [29] and real dataset YAGO [1,46] and their respective query workloads. 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. MonetDB5 11.2.0 [20] and RDF-3X version 0.3.5 were used for the experiments. 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. Only warm-cache experiments were performed for which the authors ran the queries 21 times without dropping caches, the time of the first (cold) run was ignored and the mean of the other 20 query runs was calculated and reported.

1.4.1 Description of Datasets and Query Workload

Tsialiamanis et. al [40] were able to scale SP2Bench [29] synthetic data only up to 50M triples since RDF-3X was not able to load bigger datasets1. In order to load the YAGO dataset in MonetDB some modifications had to be manually performed: invalid characters contained in YAGO's URIs (e.g., <, >, etc.) that the Redland Raptor parser does not accept were removed. In addition, RDF-3X ignores the base URI and consequently cannot distinguish between URI <abc> and literal "abc". All literals of the original YAGO dataset were therefore converted to URIs using as prefix the base URI of the corresponding RDF-XML file. By removing duplicate triples, the obtained dataset contained 16M distinct triples. The modifications were necessary to ensure that both systems yield the same results for the same query.

The study of datasets (for a complete report see [35]) confirmed the proposed optimization heuristics that were previously discussed. Regarding HEURISTIC 1 the observation that given a specific value for a subject and object, there are only few properties that satisfy the specific triple pattern was confirmed. In addition, for given values for property and object in a triple pattern, a small number of subjects that satisfied it was obtained. Similarly, it was observed 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.

The findings of Tsialiamanis et. al.[40] were also verified by the study reported in [8]. In the case of HEURISTIC 2, it was observed that join pattern p ⋈ o returns always zero results making it the most selective one. The same is true for join pattern s ⋈ p for the SP2Bench dataset, but not for the YAGO dataset (the only dataset where a URI can appear both as the property and subject of triples). Join pattern s ⋈ 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 ⋈ p yields results that are 1 to 2 orders of magnitude larger than s ⋈ s and o ⋈ 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, six conjunctive queries (and variations thereof) from SP2Bench and four queries from YAGO benchmarks were chosen. The queries used are presented in the Appendix of this Deliverable. 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*. In Table 2, **#TPs** designate the number of triple patterns, **#Vars** and **#PVs** the number of *variables* and *projection variables* of the query respectively. **#SVs** is the number of shared variables and **#TP0, #TP1** and **#TP2** denote the number of triple patterns with zero, one and two constants resp.. **#Joins** indicate the number of joins in a query. **MaxStarJoin** denotes the number of triple patterns that participate in the query's largest star join.

Tsialiamanis et. al [40] considered join queries with 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) as can be seen in Table 2. 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 SP2Bench 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 the heuristics proposed by Tsialiamanis et. al. [40] proved to be quite effective for queries whose triple patterns exhibit syntactical dissimilarities, i.e., they have different number of constants (and/or shared variables) that are found in different positions. The authors observed that the majority of the queries for both datasets considered s ⋈ s joins (suggesting star-shaped joins on the *subject* component of the triple pattern), followed by s ⋈ 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.

Query	**SP1**	**SP2a**	**SP2b**	**SP3(a,b,c)2**	**SP4a**	**SP4b**	**SP5**	**SP6**	**Y1**	**Y2**	**Y3**	**Y4**
#TPs	3	10	8	2	6	5	1	1	8	6	6	5
#Vars	2	10	8	2	5	5	2	1	6	4	7	7
#PVs	2	1	1	1	2	2	2	1	2	1	1	3
#SVs	1	1	1	1	5	4	0	0	4	3	3	4
#TP0	0	0	0	0	0	0	0	0	0	0	2	3
#TP1	1	9	7	1	4	3	1	0	6	3	2	0
#TP2	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
MaxStarJoin	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 SP2Bench and YAGO

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, Tsialiamanis et. al [40] estimated their cost using the cost model of RDF-3X [22]. In particular, the focus was 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, Table 3 does 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 were estimated using the following CDP formula:

cost_mergejoin(lc, lr) = lc+rc / 100,000 cost_hashjoin(lc, rc) = 300, 000 + lc/100 + rc/10

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

	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 for SP2Bench and YAGO Queries

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												
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	B	B	B
Similar Plans	yes	no	no	yes	yes	no	yes	yes	no	no	yes	no

Table 4: Plan Characteristics for SP2Bench and YAGO

1.4.2 Query Plans

As can be seen in Table 4 for all the queries of the workload, 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 SP2Bench 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 SP2Bench, 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 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 ⋈ s). On the other hand, for the light star query SP1 of SP2bench that involves two s ⋈ s joins, HEURISTICS 3 and 4 are used to determine join ordering, as well as the ordered relations on which selections are evaluated.

SP2Bench query SP4a and YAGO query Y3 are to a great extent syntactically dissimilar. SP4a contains small chain joins whereas Y3 contains small star joins on variables found in different positions (s ⋈ s and s ⋈ 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 CDP3 (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.

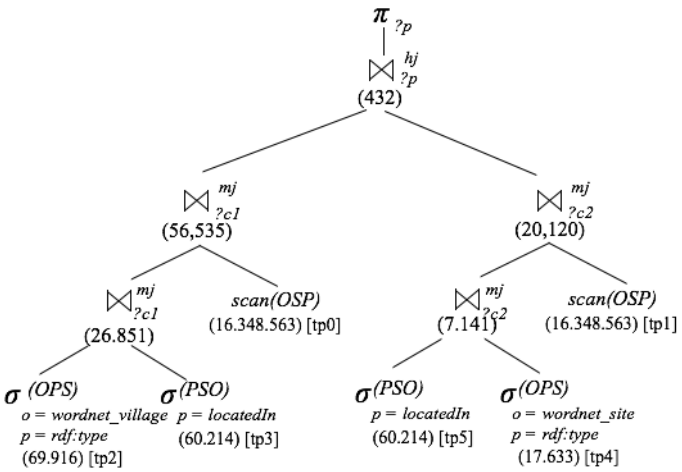

Figure 2: HSP Plan for YAGO Query Y3

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 ⋈ s with very similar triple patterns (only HEURISTIC 3 is applied). HSP 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 sub-plan thus resulting in less intermediate results. In YAGO query Y2, HSP chooses to perform all the merge joins on one variable producing a left deep plan (see Figure 3(b)), whereas CDP produces a bushy one that reduces the size of intermediate results early in the plan (see Figure3(a)).

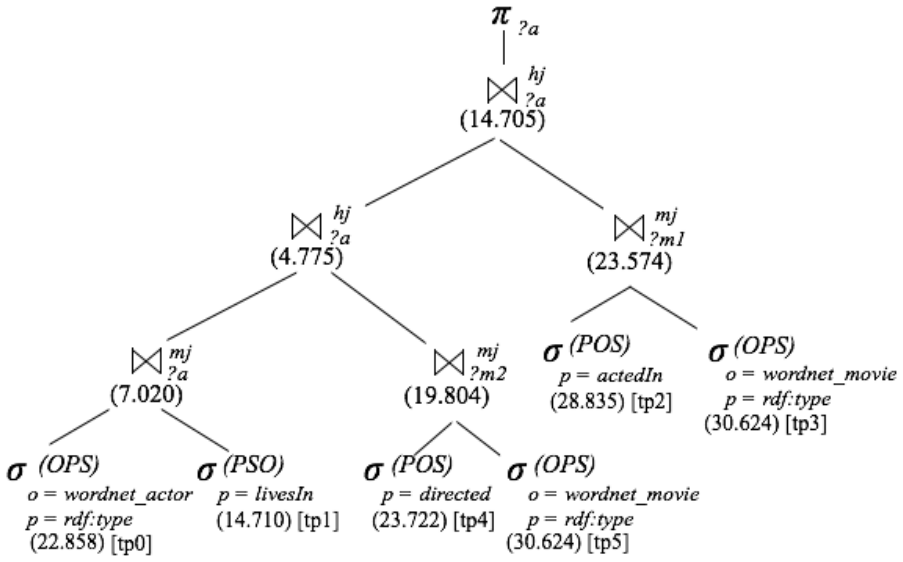

Figure 3(a): CDP Plan for Query Y2

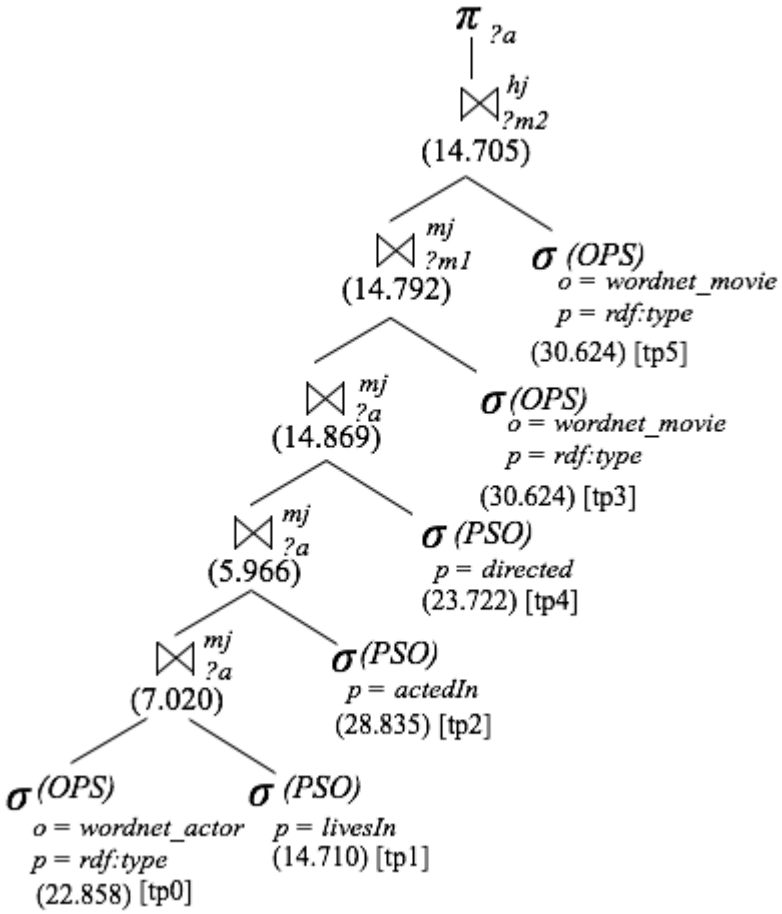

Figure 3(b): HSP Plan for YAGO Query Y2

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.

Last, the authors also compared the efficiency of the plans obtained by the SQL translation of the SPARQL queries in the workload. Those served as the baseline experiment for the plans produced by the standard MonetDB/SQL optimizer. Since unlike HSP and CDP, the MonetDB/SQL optimizer produces only left deep plans, the SPARQL queries were not directly translated into SQL ones using exactly the same access paths as those employed by the HSP plans. The choice was 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, the ordered relation was chosen according to HEURISTIC 1. Thus, the MonetDB/SQL optimizer will undertake the task of join ordering using runtime optimization techniques (e.g., sampling).

1.4.3 Query Execution Times

Tsialiamanis et. al [40] discussed 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. They additionally reported the execution time of its equivalent SQL rewriting when evaluated by the standard MonetDB/SQL optimizer.

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)

Table 6 shows the time needed for the construction of HSP plans, without considering query evaluation. The planning times for the HSP are very short (between 100 and 200 microseconds). Moreover, it is logical to expect that 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.

	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	-	1,909.13	0.09	0.48

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

	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)

The execution times the experiments conducted for the SP2Bench 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 the reported 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. Note also 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 MonetDB/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.

2. RDF Clustering and Indexing

As part of the LOD2 project, research is ongoing (2011-2014) at CWI in RDF/graph data clustering and indexing. With the availability of MonetDB+HSP, there is now an initial platform available for experiments. A first step has been to evaluate data locality, shifting the focus to **analytical RDF queries**, concentrating on physical data clustering. We note that both the SP2Bench and the Yago queries used in the evaluation of HSP in Section 1 of this deliverable consist mostly of lookup and navigational queries; hence more mimic a lookup-intensive workload. Even if there are joins, the data volume selected by the queries is typically small, and these queries lack grouping and aggregation. Even though RDF systems initially have been used mostly in interactive web applications leading to such a lookup-intensive workload, the consolidation of RDF data in warehouses that leverage its ability to query over data from many different sources implies the need for supporting more analytical workloads as well [55]. The needs for analytical query processing SPARQL, most notably nested subqueries and grouping+aggregation, has been addressed in SPARQL 1.1, which is at the time of this writing is close to becoming a W3C recommendation. Many SPARQL systems already support these features. The database group of CWI is rather specialized in analytical workloads and therefore this is the topic of our future research approach.

This section is structured as follows:

Section2.1: we examine data locality in RDF more closely, and sketch our proposal for Characteristic Set Clustering of RDF data. This can help create more locality in selection queries, even accross joins between multiple Characteristics Sets. It also enables the development of more CPU efficient SPARQL query processing algorithms (RDFscan/RDFjoin), strongly reducing the amount of joins.

Section 2.2: we conduct experiments with these ideas on the RDF-H benchmark with the MonetDB RDF+HSP prototype, showing that clustering and RDFscan/RDFjoin and can strong benefit RDF analytical workloads.

Section 2.3: we sketch a research roadmap to elaborate on these ideas. Apart from clustering on characteristic sets, we also propose on-the-fly (runtime) value and path indexing techniques, as opposed to static (at bulk-load) indexing.

These ideas will be further worked out and implemented in the remaining two years of the LOD2 project. In the Appendix (Section 4.3) one can find a technical blue-print of a new MonetDB RDF support module that incorporates the above ideas.

2.1 Motivation

A crucial aspect of efficient analytical query processing is data locality. The current MonetDB RDF storage still adopts the classical approach of storing the Subject (S)-Predicate (P)-Object (O) triples in all possible six orders: SPO, SOP, PSO, POS, OPS, OSP. At first glance, this abundance of access paths might seem to create good access locality – later more on that. The HSP front-end tries to maximize the span of query subgraphs that are covered by a single order, and in which the self-joins against the triple table can be handled by efficient merge-joins. However, even relatively simple SPARQL queries typically cannot handle all self-joins with merge-joins. This then leads to random access patterns and loss of locality. The damage comes in the form of memory-hungry hash-joins that cause a random access pattern (ill supported in modern hardware) and potentially also large table scans. As such, on analytical workloads, we find that RDF stores have little ability to obtain data locality, in contrast to relational equivalents. Top relational systems in TPC-H such as Vectorwise [47] employ clustered indexing strategies that benefit the main range selection predicates (typically on date-related selections) and additionally get merge-join like locality on the most important foreign key joins. Such locality is currently unattainable in any RDF store, as the concept of clustered indexing is missing. In effect, RDF systems always answer queries from a combination of unclustered indexes.

One might remark that in spite of this (perceived?) disadvantage, RDF has the advantage that it can query

much less structured data that that corresponding to the relational model. That is certainly true. It is however a missed opportunity in the face of that fact, to therefore act as if RDF data had no structure at all. From a relational database technologists' point of view, RDF is just a data format and data represented in it will have just as much structure as when that data would have been represented in a different format. In real-world RDF this structure typically surfaces as follows (.. as **correlations**):

- certain kinds of subjects have the same set of properties (belong to the same class)
- certain classes are connected, over the same kind of property paths (foreign key relationships)

Of course, in addition to these regularities, real-world RDF data also contains irregularities caused by the presence of many such schematic structures inside a same dataset, data dirtiness leading to inconsequent usage of e.g. property URIs, missing data and duplicate data.

Exploiting structure in a database system can achieve better locality, but there is a second opportunity here to speed up SPARQL query processing: removal of unnecessary joins. SPARQL query executors that lack the certainty that a certain join, e.g. getting an object value over a property using a subject as lookup key (part of the typical star pattern), will produce **exactly** one result. In general, a join may produce 0 results, in which case the entire set of variable bindings should be removed from the query result bindings, or even produce multiple results, in which case for each result a set of bindings should be duplicated. Thus, each SPARQL pattern leads to a separate join (in the best case, a merge-join). However, relational query processors that know about the structure of the data waste no effort here. The only joins they process are "real" joins between different entities. Henceforth, we pose that if RDF storage could recognize the latent structure in datasets, parts of the join overhead in SPARQL could be avoided. This is a second opportunity, besides the data access locality opportunity, that we pursue.

2.1.1 Characteristic Sets

The important paper, which we build on is the so-called "Characteristic Sets" paper [21]. This paper shows that it is relatively easy to recover large part of the implicit class structure underlying data stored in RDF triples. The idea is simply to keep track of characteristic sets of properties that occur frequently with the same subject. A characteristic set thus is a set of property URIs, combined with a measure of support (e.g. the amount of subjects in the dataset that have this combination of properties). Characteristic Sets can be found quickly using an APRIORI like algorithm [54] or even more simple using a lossy counting histogram in a single pass with little memory requirements.

Even in web crawled RDF, which is considered the dirtiest data encountered in practice, there is still quite a bit of regularity that characteristic sets bring to light [21]. The irregularity in it typically surfaces as:

- missing properties that would have been expected (incomplete data),
- multi-valued properties (more than one triple with same subject and property),
- unexpected properties that often occur at low frequency (properties that seem out of place for a subject, or properties that occur very infrequently in the dataset -- often spelling errors) .

On a high level, each characteristic set could be seen as some kind of class definition such as RDFS could have provided. Of course, RDF data may not fully conform to these hypothetical class definitions due to incomplete, dirty or simply highly variable data, and it is common to store data in RDF that is much less structured than typical relational data. However, the characteristic sets do not have to describe 100% of the triples in a dataset; as long as the great majority of the triples are described by them, most of the benefits that we will aim at will be obtained.

In [21] the characteristic sets are developed and used as a means for achieving better cardinality estimation in SPARQL query optimization. For example given schematic knowledge about Books (Book=some characteristic set), e.g. the presence of a 'ISBN' property makes the co-occurrence of a 'Author' property almost a certainty, thus affects query result size estimation. Assuming that the dataset holds many different kinds of data, normally one would estimate the occurrence of Author properties for a given subject much lower. Hence, an important source of correlations that hinder SPARQL query optimization (as described in

Section 1.1) in the context of the HSP front-end, can be improved by keeping knowledge of Characteristic Sets as optimizer statistics. In an unpublished presentation, the authors of [21] have stated that even though the quality of query cost estimates improved greatly, the overall benefit in their RDF-3X system in terms of query performance of such improved knowledge they have seen is limited. This might seem a contradiction. We argue here that knowledge alone about structure is not enough; storage should exploit this knowledge to achieve better access patterns. Thus, without accompanying improvement in RDF storage an indexing, significant performance improvements will be hard to come by—hence our focus on **Subject Clustering** of RDF. Further, once we have RDF storage that has become aware of characteristic sets appearing in the data, this regularity could also be exploited during query processing by reducing the amount of SPARQL join effort, by deriving new RDF scan and join primitives (**RDFscan**, **RDFjoin**) that e.g. retrieve complete start patterns at once, exploiting characteristic set information.

As an aside, we will digress into the ease of use of SPARQL in the face of data diversity and dirtiness Querying highly unstructured data with SPARQL is difficult; as the user posing almost any query must rely on (implicit) knowledge of how data is structured, and SPARQL does not offer specific features that make querying data with unknown or variable structure any more easy than in the relational case. On the contrary, in the relational case, a user could at least read the schema or browse it in a schema designer GUI, which are commonly available components in any system. As much RDF data does not have an explicit schema, such a facility is typically absent in RDF stores. Therefore, an RDF system that automatically derives the characteristics sets of the RDF data it stores and the ("foreign-key") inter-relationships between them, provides the potential for an important feature for SPARQL users: the system could DESCRIBE the general structure of the data set in textual form or via a schema GUI, summarizing the most important characteristic sets and their inter-relationships.

2.1.2 Triple Indexing to the Rescue?

As mentioned, both the current MonetDB RDF support (see Section 1.3.2) and in fact most other current RDF systems store data with triples sorted on the various permutations. We briefly discuss these orders in somewhat more detail, to more clearly argue on the measure of locality that they provide:

- SPO: for each subject we get all the properties near each other (much like NSM in relational systems). However, the records (subjects) of all different "tables" (in the relational analogy) are stored interspersed in seemingly random order.

  SPO is convenient for transactional workloads, where queries affect typically a single subject, but all its properties (insert/delete/lookup).

- PSO: an order that resembles DSM: for each property we get subjects and values. Given that certain properties often only occur for a single class of subjects, it is quite near to DSM. But not fully, as certain properties do occur for multiple subject classes, and the subjects are thus still somewhat interspersed.

  PSO is important for analytical workloads, which may visit many different subject, but only focus on a subset of the properties (thus, such queries would leave the parts of the PSO table that correspond to the irrelevant properties untouched, improving the access pattern over SPO).

- OPS and POS: similar to value indexes in relational systems. The latter is quite often used as many queries contain a known property and may filter on O, which fits POS (given a P, look for O).

  These orderings are typically used in selections to obtain an S (and sometimes also a P) from a restriction on (literal) O; or for 1:N joins (reverse foreign key traversal). Since S is in last position, such access typically leads to a set of S OIDs that is scattered all over the dataset, such that subsequent query steps into e.g. a SPO or PSO lack locality. Therefore, these are best used in transactional workloads, where the number of looked up subjects (= nr. of random I/Os) is low.

- OSP and SOP are created by exhaustive indexing schemes, but tend to be less often used, since one subject tends to have only a limited number of properties.

We remark that column store systems may store each of the three columns separately, and thus e.g. in case of PSO and POS would store two identical sorted P columns. This column could be stored only once and reused. So instead of 6x3=18 columns, an exhaustively indexing column store would store 15. Note that this is a replication by a factor 5. From a relational point of view, that much replication is quite significant, not only from the point of view of increased storage size. The main problem is that bulk load, backup/restore and most importantly update performance gets affected with a higher degree of replication.

A further remark is that by now, support in RDF products for the RDF Data Graph feature has become prevalent. Consequently, triple storage becomes quad storage consisting of a P,S,O and G identifier, and an exhaustive indexing strategy now entails 24 orderings. A column store would store 24*4=96 columns, but the columnar reuse could reduce that to 64. This is a replication factor of 16, and hence even more unattractive to carry through, especially in light of the fact that the locality benefits of exhaustive indexing are questionable. Let us elaborate on that more deeply now.

While it seems that RDF systems with exhaustive indexing would get near to the efficiency of relational database engines, this is not really the case. In fact, in the best case this indexing gets RDF systems back to un-indexed locality in relational systems, because queries will combine different patterns (SPO, PSO, OPS) and making the joins between those does not have locality. This is similar to relational query processing without index locality, i.e. unclustered index access: while the first access to the index is fast, the subsequent RID-list needs to be fetched from the main table leading to many lookups. Even if the lookups can use index structures (B-tree), we get large amounts of scattered fetches, which on current hardware does not scale. Due to the growing imbalance between latency and bandwidth, both in the disk and the RAM level of the memory hierarchy, the cut-off point where such index-lookup queries are better than sequential column (range) scans has been exponentially decreasing. Where in the 1990s a 5% selection predicate could be run efficiently on such a plan, in 2012 this is now 0.0000005% (and decreasing every year).

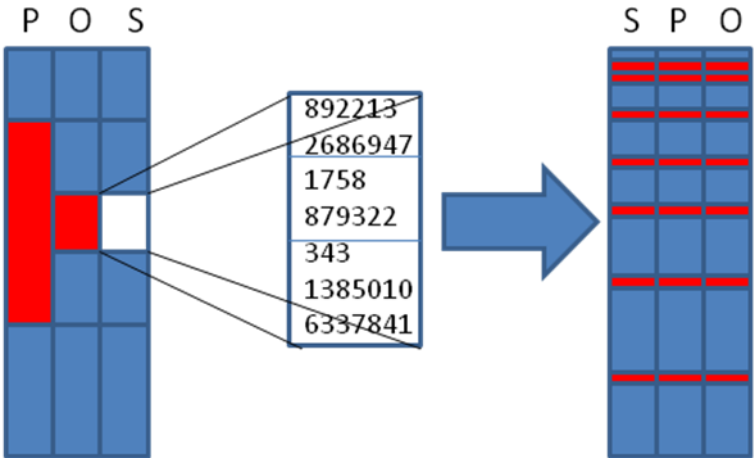

Figure 4: even merge-joining multiple triple patterns leads to scattered access (non-locality)

It is the express purpose of HSP to minimize the transition points between orderings, as it aims to maximize the amount of order compatibility and merge-joins. However, even simple SPARQL queries that have only merge joins may still run into loss of locality, as pictured in Figure 4 (more complex SPARQL queries that join multiple different subject variables lead to hash-joins also in HSP, which also destroy locality of access). Typically, SPARQL query evaluation starts with some filter or constraint that is best handled with OPS (or POS), but subsequently has to handle a star pattern for fetching additional properties of those selected subjects for which a switch to SPO or PSO is required to leverage the indices and avoid a hash-join against a full triple table. This would be the equivalent of the simple scan-select SQL query:

```
select PROP1,..PROPn from CS where PROPx between LO and HI
```

where CS stands for objects of a certain class or, in our context, a characteristic set (“CS”). It is good to remember that a relational database system in such a simple query pattern gets **fully clustered** data access (reading **only** result tuples) if table CS is stored as a CLUSTERED INDEX on column PROPx.

In order to support not only fast access to PROPx but also to other dimensions, modern relational database systems also offer multi-dimensional indexing techniques such as MDX in DB2 [50], or the possibility to replicate data in multiple such clustered orders, using materialized views or column projections [49]. Such techniques are quite important in the efficient handling of large data warehouses with snowflake-like relationships. Until now, **RDF systems have nothing to offer in this department** since despite their exhaustive indexing on all orders still fall into the trap of non-locality of access.

2.1.3 Subject Clustering

There is rarely an RDF system that operates directly on URIs as strings; typically these URIs are mapped to more compact and cheap-to-manipulate integers first (and in some cases, a similar mapping to integer object identifiers is made for literals). The most common approach, which is data-import friendly, is to simply assign OIDs in the order data is parsed. We notice that this simple approach is also followed by the current URI dictionary used in the MonetDB RDF module, described in Section 1.3.2.

We argue that an issue typically overlooked in RDF storage is the way how to obtain S/P/O object identifiers (OIDs) from URIs or literals. Assigning OIDs for S,P,O in data import order might be quite random and uncorrelated with the access paths of interest to the database users. Given the fact that the OID order (whatever it happens to be) is heavily exploited in RDF systems, this is in fact the direct cause of non-locality in RDF query plans.

To get real locality we would like to order the OIDs in a meaningful way. Let us discuss S, O and P identifiers in turn. For S object identifiers we should:

- group them by characteristic sets: in a RDF-H example (see later) we would first we assign all OIDs to subjects of ORDER, then the following OID range to subjects of LINEITEM, etc. (ORDER and LINEITEM are both characteristic sets)
- within a characteristic set, we can then further sub-order them on some index keys (i.e. property values). An extreme form of this is to adopt a multi-table clustering strategy for this ordering, such as Bitwise Dimensional Co-Clustering (BDCC) [44].

Similarly, the O OIDs used for literals should be ordered in a way that is meaningful to SPARQL value comparison semantics, such that comparisons on the O identifiers can be used for executing value range-predicates. The MSc thesis [43] describes a literal-to-string mapping that respects SPARQL comparison semantics, and thus allows a query optimizer to compile SPARQL range-predicates on literal values onto range-predicates on O identifiers. Note that not all RDF systems use OIDs to represent literal values, but implement a special RDF "any-type" that can contain literals of any type. In this case, it again pays off to have the binary representation of that type be organized such that SPARQL semantic compliant range-comparisons can be executed efficiently on the binary representation of the literals.

Regarding the choosing of P identifiers, there are typically few values P that are very frequent. But there may be a long tail of infrequent P's (often spelling errors or low-value noisy properties). Such P's could be treated differently than the frequent P's.

The focus of our work here is the first topic: the ordering of the subject -OIDs, which we call **Subject Clustering.** We notice that the game of manipulating the order of the OIDs of the nodes in the labelled graph formed by the subjects and properties connecting them amounts to **graph clustering**. This, because in the end the OID values determine the physical location of a triple inside a triple table that is kept in some order. This physical location is a place in memory, and on disk, and in an MPP system across nodes in the cluster of the graph objects. The graph clustering we are suggesting here lets this order be determined by the structure of the node (e.g. the CS to which a subject belongs) and further by the node property values one or multiple hops away. That is, when considering subject OIDs, it is the value of an object one property away (if we cluster on a local property of the characteristic set) or even multiple hops away (if we cluster over “foreign keys”). If we adopt a multi-dimensional indexing approach, it will be even a combination of multiple such paths of the graph that will determine the physical placement of the node.

2.1.4 RDFscan and RDFjoin

It is well-known that SPARQL queries lead to query plans with many join operators; typically one operator per SPARQL variable pattern. If one thinks about the equivalent SQL query of any SPARQL query, reformulated in terms of tables corresponding to characteristics sets, such SQL query have much less joins, typically the amount of different characteristics sets involved minus one.

The many self-joins of current SPARQL processors pose a data access problem, which HSP (and other systems) try to tackle by choosing the best possible access pattern. This means that in HSP, the typical star-pattern in SPARQL queries centered around a common subject, is handled by a sequence of merge-joins. However, in case of mostly regular data, these merge joins in the end have no work to perform as data from the various ordered input streams tends to be lined up pretty well. If the data is really regular, all "column" values (the property bindings) for the various properties in the star come exactly aligned from the various merge join input streams. This pattern is the same as column-store systems use to scan multiple columns; it is widely recognized that this operation of "stitching together" columns has a much lower CPU cost and complexity than real joins (its complexity is linear in input, not quadratic as joins in worst case are) [2,3].

The core idea of our novel RDF storage proposal (see Section 4.5) is to store RDF data that has been recognized as conforming to a characteristic set together in an aligned way, such that for a whole stretch of subjects we get aligned stretches of Objects; for each "exactly one" property in the Characteristic set. Triples that do not conform to a characteristic set, are stored separately. This new storage layout can be exploited by the new operator **RDFscan** that delivers a tuple stream for multiple properties in one go. A slight variant is the RDFjoin, which does the same, but receiving a stream of candidate subjects. The latter operator was recently proposed and termed "Pivot Index Scan" [42]. In contrast to this previous work, our RDFscan/RDFjoin work in conjunction with our proposed characteristic set storage of the RDF data, and to exploit this: if multiple properties that need to be delivered come from one characteristic set, we can exploit the fact that the Object values are stored aligned and can deliver the results without CPU join overhead.

The RDFscan/RDFjoin can also combine with the Zone Map idea (see next section for a description of Zone Maps): if a query just contains a bound Property-Object pattern, that basically is there to filter the results to those subjects that have such a triple. A typical example is testing RDFS membership (e.g. ?subj a <LINEITEM>). If we know that "a" is a property that is part of the characteristic set we are scanning, we can use a ZoneMap index on that property to check whether a whole stretch of triples conforms to this predicate in one go. In that case, testing for this predicate can be omitted from further RDFscan/RDFjoin processing.

In all, we think that storage of triple data around the concept of characteristics sets has great potential to reduce CPU effort in SPARQL query processing.

2.2 MonetDB HSP Experiments

To test whether our ideas have any merit, we conducted initial experiments on the RDF-H benchmark (See: sf.net/projects/bibm, in the tpch subdirectory), where we compare a normally loaded version of the benchmark dataset with a clustered version, where all triples of the two main object classes (ORDERs and LINEITEMs) get subject OIDs that reflect the "orderdate" of the ORDER to which they belong.

2.2.1 RDF-H

The RDF-H benchmark is a straight 1-1 mapping of the TPC-H benchmark to SPARQL. As such, we recognize that this benchmark is quite relational, still the quality of TPC-H as an analytical benchmark is widely recognized, and initial experiments performed by Openlink (who developed RDF-H in the context of the LOD2 project) have indicated that RDF-H heavily stresses the state of the art in analytical query performance of RDF systems. As such, this turned out to be a more challenging benchmark than the BSBM-BI (Business Intelligence) benchmark, which is currently the only analytical RDF benchmark alternative. We re-iterate that the SP2Bench and Yago workloads using in the evaluation of Section 1 are mostly point-lookup and small navigations; and are not really analytical queries. Hence our choice for RDF-H.

Additionally, in turning towards analytical workloads, one has to use SPARQL1.1 rather than SPARQL1.0 since the latter does not support sub-queries, group-by aggregates or even counting. As such, the current state of HSP, which supports only the SPARQL1.0 was a limitation. This was dealt with by choosing TPC-H queries without complex (correlated, or aggregating) subqueries. Further, the top-level group-by/order and aggregation was removed from the queries, noting that TPC-H query performance tends to be dominated either by scanning I/O or join performance (not by the top-level group-by).

In TPC-H, the single largest class of objects are so-called LINEITEMs. In the SF=1 dataset size, there are 6 million such objects. In our experiments, we use the SF=10 dataset, such that there are 60M. Each LINEITEM has 16 properties – given that the data generator of RDF-H is based on the relational TPC-H data generator this is perfectly regular. The second most important class of objects are ORDERS, of which there are 1.5M at SF=1 (and hence 15 million in our SF=10 experiment). Orders have 9 properties, and in RDF-H each LINEITEM is linked to exactly one ORDER using the "has_order" relationship property (a "foreign key"). Each ORDER has 1-8 LINEITEMs, and on average exactly four.

There are further tables in TPC-H, corresponding to RDFS classes in RDF-H, namely CUSTOMERs (each ORDER comes from one CUSTOMER) and SUPPLIERs (each LINEITEM is a PART provided by one SUPPLIER). There is the PARTSUPP class that describes which PARTS are provided by which SUPPLIERs. Finally, both CUSTOMERs and SUPPLIERSs are in a NATION, and each NATION belongs to a REGION. In SF=1, there are .8M objects in PARSUPP, .2M objects in PART, .15M objects in CUSTOMER, and .01M objects in SUPPLIER. NATION and REGION are fixed-size dimension tables.

The total size of the RDF-H dataset at SF=10 (i.e. ASCII raw data size of the relational CSV format of 10GB) in MonetDB RDF is 190GB. There are 1.25 billion triples in total in the dataset; and the naive MonetDB storage now uses 18 columns of 8 bytes (1.25*18*8=180GB; with 5GB in the URI dictionary and 5GB in the literal map). The reckless exhaustive indexing, uncompressed data storage and absence of leading column sharing in the current MonetDB RDF support module causes this high space consumption of 152 bytes per triple. In contrast, Virtuoso v7 which features compressed column storage, and stores 14 columns (POGS, PSOG, OP, SP, GS) needs just 10 bytes/column [47], measured on DBpedia. The older, also compressing row storage of Virtuoso 6 holding the same column combinations needs roughly 30 bytes/triple. The main point of the RDF-H experiments in MonetDB are not these data sizes, but a comparison between the access pattern of a naive OID-in-parsing-order approach versus a graph-clustering-OID assignment approach in the same system. Regardless the current storage inefficiency of MonetDB RDF, the **relative** access footprint **savings** will translate into any RDF triple indexing storage scheme.

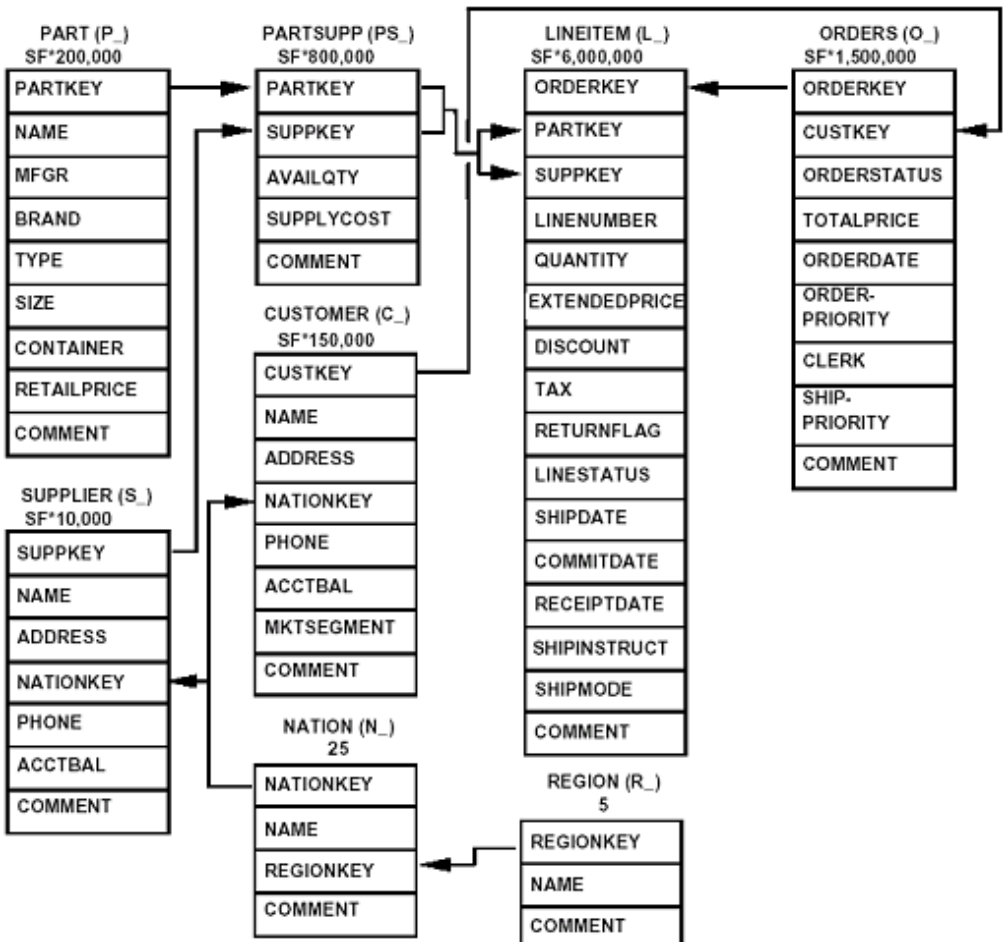

Figure 5: TPC-H relational schema

Below we show a snippet of the LINEITEM triples in RDF-H:

```
rdfh-inst:lineitem_1_1
    a rdfh:lineitem ;
    rdfh:has_order rdfh-inst:orders_1 ;
    rdfh:has_part rdfh-inst:part_15519 ;
    rdfh:has_supplier rdfh-inst:supplier_785 ;
    rdfh:linenumber 1 ;
    rdfh:linequantity 17e0 ;
    rdfh:lineextendedprice 24386.67e0 ;
    rdfh:linediscount 0.04e0 ;
    rdfh:linetax 0.02e0 ;
    rdfh:returnflag "N" ;
    rdfh:linestatus "O" ;
    rdfh:shipdate "1996-03-13"^^xsd:dateTime ;
    rdfh:commitdate "1996-02-12"^^xsd:dateTime ;
    rdfh:receiptdate "1996-03-22"^^xsd:dateTime ;
    rdfh:shipinstruct "DELIVER IN PERSON" ;
    rdfh:shipmode "TRUCK" ;
    rdfh:comment "egular courts above the" .
rdfh-inst:lineitem_1_2
(etc)
```

We see that rather than working with explicit foreign keys, objects refer to each other using a relationship property. In this case, the l_orderkey foreign key reference part of the schema has been substituted by the

has_order property that directly refers to an ORDERs subject:

```
rdfh-inst:orders_1
    a rdfh:orders ;
    rdfh:orderkey 1 ;
    rdfh:has_customer rdfh-inst:customer_3691 ;
    rdfh:orderstatus "O" ;
    rdfh:ordertotalprice 194029.55e0 ;
    rdfh:orderdate "1996-01-02"^^xsd:dateTime ;
    rdfh:orderpriority "5-LOW" ;
    rdfh:clerk "Clerk#000000951" ;
    rdfh:shippriority 0 ;
    rdfh:comment "nstructions sleep furiously among" .
rdfh-inst:orders_2
(etc)
```

2.2.2 RDF Clustering Scheme on orderdate

We modified MonetDB RDF storage and the HSP front-end to enable RDF clustering experiments. The effects of our changes come down to the following three modifications:

- We recover the characteristic sets (aka, the "table schemas") of an imported RDF dataset, and the foreign key links between them (via the connecting properties like "has_order" for LINEITEM objects) in the RDF-H dataset. In this case, the characteristic sets cover 100% of the data, as RDF-H is fully relational. We note that it is not our assumption that this would hold in general. Our new MonetDB RDF storage proposal in Section 2.3 also works in situation where not all subjects pertain to a characteristic set.

- We re-arrange the data storage with the CS knowledge, identifying a clustering order for each CS. This is currently done manually, but can become (semi-) automatically given hints about relevant index properties. These would then be used for the CS that holds such a property, and be imported to those CS's that do not hold them, but can reach them via relationships [44].

  In this case, this meant we created clustered storage for the ORDER objects on "orderdate", and clustered storage for the LINEITEM objects on orderdate as well, as reachable over the "has_order" property. This means that the ORDER subjects are given out incrementally in orderdate order, such that OID order respects orderdate order for ORDERS object. Subsequently, we give out LINEITEM subject OIDs in a closely similar order corresponding order. For this purpose, we attach the clustering criterion (orderdate) also as a "virtual" property to the LINEITEM subjects.

- We exploit *selection pushdown over relationships*. When a query contains a range-selection predicate on orderdate and joins with LINEITEM properties, it is possible to push down the selection (i.e. fully avoid I/O and selection processing) on both the ORDERS and LINEITEM column (property) scans; because the subject identifiers of both characteristic sets conform to orderdate order and thus only a sub-range of all subject-OIDs is known to contain relevant triples. This can be realized with a a well-known analytical storage optimization structure, which is called **"ZONE MAPS"** in the Netezza product and called MinMax indices in Vectorwise [48]. The idea is to keep for each column (here: property) of a table (here: characteristic set) the minimum and maximum value, automatically. Rather than a global MinMax per column, it is kept for each subsequent set of say 10.000 tuples. A 60M tuple table thus yields 6000 such MinMax values for each range of 10.000 consecutive tuples (triples).

  The idea behind such indices is to automatically detect and exploit value correlations. There are many such attribute correlations in real life, and a typical example are in fact date attributes. The lifetime of a product from ordering to being delivered vary between a few days and at most a few months. This means that the difference between a l_shipdate property of a LINEITEM and the o_orderdate of its corresponding ORDER is at most (say) three month away. Given that the subjects

in this setup will be clustered on o_orderdate, the subject order will also show a strong correlation with l_shipdate. Consequently, l_shipdate selection predicates can also be used to push down selections on LINEITEM scans.

Finally, when considering a scan on LINEITEM subjects in a query that connects to ORDERs and has a restriction on o_orderdate, we can exploit also the zone map on the "virtual" o_orderdate attached to LINEITEM to also restrict the scan on LINEITEM if there is an o_orderdate predicate. Additionally, if there would be a predicate not on o_orderdate but on l_shipdate, it is possible to use the Zone-Maps on both l_shipdate and the "virtual" orderdate to derive a Min and Max bound on o_orderdate from a restriction on l_shipdate (as these are correlated, it will be a limited span of orderdates). This derived o_orderdate bound can subsequently be used by the query optimizer to also restrict the ORDERS scan -- even if the query does not have an o_orderdate predicate at all.

As such, the benefit in the clustered scheme is that all selections on any kind of date property push down on all scans. These kind of techniques are the mainstay of relational data warehousing performance and if SPARQL systems want to compete there they need to smarten up.

2.2.3 RDFscan/RDFjoin

The fact that the RDF-H data is really regular enabled us to quickly experiment with the ideas underlying RDFscan and RDFjoin. That is, on the RDF-H dataset even the default MonetDB RDF shredder will currently produce highly regular tables in the PSO order. This data is highly similar to relational column-store layout: DSM, as for each property (i.e. column) it contains all object bindings in order.

Further, the bulk loading method of RDF-H first presents all LINEITEM triples, then all ORDERS triple, etc. If the table would be bulk-loaded in order from small dimension tables to bigger and finally the huge LINEITEM fact table, than the parse-order URI assignment scheme would even produce consecutive Subject OIDs as well. However, we did not exploit that here.

What we did exploit is:

- the properties occurring in the RDF-H benchmark are all "exactly one" properties: each subject belonging to a certain characteristic set (table) has exactly one triple binding an object value to each of the properties of the characteristic set. This means that if we look at the subsequences in the PSO table of all these properties, the S column is identical among them, and all O values are aligned. MonetDB being a column store, this means that these S column slices are aligned columns, as if this were a columnar relational database. Thus, we can skip the whole phases of joining on the S-OIDs when performing an RDFscan.
- all subjects from the same characteristic set (e.g. LINEITEM) not only have exactly one RDFS "a" property, but it also has a constant value (here: a rdfh:lineitem). This means that in Q3 and Q6 we can omit checking for the presence of these triples.

Thus, the full regularity of RDF-H allowed us to experiment with RDFscan without implementing the characteristic set RDF storage scheme fully; we could just rely on MonetDB RDF+HSP for this. On the flip side, the results presented here presented the absolute best case where 100% of the data was characteristic set conformant. In real-life data, there will be a (minority) percentage of triples which do not belong to a characteristic set. The RDFscan will also have to access such triples using a more costly code path. However, we conjecture that typically, the most regular data is often also the most queried data, as formulating queries over irregular data is difficult (e.g. think of the long tail of highly infrequent property names; often representing properties with typo's in them -- these are unlikely to get queried as SPARQL users will likely not formulate queries that contain all possible spelling errors of the property URIs that they find interesting).

QUERY PLAN	STORAGE SCHEME	ZONE MAPS (selection pushdown over relationships)	Q3 (select-join-select) COLD	Q3 HOT	Q6 (scan-select) COLD	Q6 HOT
Default (merge joins)	Parse order	no	37.50	19.66	28.25	6.52
	Clustered (orderdate)	no	18.01	15.32	9.27	3.27
		yes	2.13	2.02	n.a. (one relationship)	
RDFscan/ RDFjoin	Parse order	no	3.34	2.93	8.64	2.16
	Clustered (orderdate)	no	2.13	2.01	1.47	0.44
		yes	0.89	0.78	n.a. (one relationship)	

Table 9: MonetDB RDF+HSP performance on RDF-H (SF=10), using various optimizations (time in seconds)

Subsequently, we present some experiments on a Linux machine with two CPUs (16 cores, running at 2GHz) of the Sandy Bridge architecture, coupled with 256GB RAM and two magnetic hard drives in RAID (180/MB/s sequential throughput).

2.2.4 Exhibit 1: RDF-H Query 6*

TPC-H Query 6 is rather simple: a single table scan with a global computed sum:

```
SELECT SUM(L_EXTENDEDPRICE*L_DISCOUNT) AS REVENUE
FROM   LINEITEM
WHERE  L_SHIPDATE >= '1994-01-01' AND L_SHIPDATE < '1995-01-01' AND
       L_DISCOUNT BETWEEN .05 AND .07
```

Note that the TPC-H dataset consist mostly of LINEITEMS (96 million out of 125 million triples) so this query scans through most of the data (albeit only touching on 5 of the 16 properties of LINEITEM). Further, the selection percentage on the discount predicate is roughly 33%. More importantly, the 1-year range selects 16% of the data (as TPC-H contains 6 years worth of data warehouse facts). This predicate is more important that the previous two (which amount to a similar 15% selectivity) since the latter one can be pushed down into the scan.

We adapted Query 6 to exclude the top-level aggregate in order to rewrite it into a SPARQ1.0 expression that performs the core of the original job yet is runnable on MonetDB RDF + HSP.

```
select
  ?l_lineextendedprice, ?l_linediscount
where {
    ?li a rdfh:lineitem ;
       rdfh:l_lineextendedprice ?l_lineextendedprice ;
       rdfh:l_linediscount ?l_linediscount ;
       rdfh:l_linequantity ?l_linequantity ;
       rdfh:l_shipdate ?l_shipdate .
    FILTER (?l_discount >= "0.05e0").
    FILTER (?l_discount <= "0.07e0").
    FILTER (?l_shipdate > "1994-01-01").
    FILTER (?l_shipdate < "1995-01-01").
}
```

It is rather by accident that the MONETDB RDF + HSP prototype can execute the comparisons on shipdate and discount, since it currently only treats strings as literals. But due to the regular syntactical format of the RDF-H data, this query produces the expected result.

In our experiments, we compare the performance of the standard MonetDB RDF+ HSP storage with orderdate clustering on LINEITEM (the only kind of data touched by Q6). This query has a one-year restriction on shipdate, which correlates with orderdate. Therefore, clustered plans can push-down this selection saving almost a factor 4 of I/O volume, which leads to roughly a factor 3-4 improvement. This benefit of clustering holds both for the classic query plans based on merge joins, as well as for RDFscan powered plans. The combined benefit of both optimizations is a factor 20 on cold data.

2.2.5 Exhibit 2: RDF-H Query 3*

Our second experiment highlights a query that includes a heavy join.

```
SELECT L_ORDERKEY, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE, O_ORDERDATE, O_SHIPPRIORITY
FROM CUSTOMER, ORDERS, LINEITEM
WHERE C_MKTSEGMENT = 'BUILDING' AND C_CUSTKEY = O_CUSTKEY AND L_ORDERKEY = O_ORDERKEY AND
O_ORDERDATE < '1995-03-15' AND L_SHIPDATE > '1995-03-15'
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
ORDER BY REVENUE DESC, O_ORDERDATE
LIMIT 10
```

The RDF version of this query we used removes the grouping, which in this case could be a heavier operator since it is grouping effectively by order. Still, in many database systems, including Vectorwise and SQLserver, this grouping takes only a minority of the time (<10% see for a SQLserver profile www.qdpma.com/tpch/TPCH100_Query_plans.html), and most of the time is spent joining and selecting, which is our focus here. The RDF query we used reads as follows:

```
select
  ?o_orderkey, ?o_orderdate, ?o_shippriority, ?l_extendedprice, ?l_discount,
where  {
  ?li a rdfh:lineitem ;
    rdfh:l_lineextendedprice ?l_lineextendedprice ;
    rdfh:l_linediscount ?l_linediscount ;
    rdfh:l_has_order ?ord ;
    rdfh:l_shipdate ?l_shipdate .
  ?ord a rdfh:orders ;
    rdfh:o_orderdate ?o_orderdate ;
    rdfh:o_shippriority ?o_shippriority ;
    rdfh:o_orderkey ?o_orderkey ;
    rdfh:o_has_customer ?cust .
  ?cust a rdfh:customer ;
    rdfh:c_mktsegment "BUILDING" .
  filter (?o_orderdate < "1995-03-15").
  filter (?l_shipdate > "1995-03-15")
}
```

This query asks for all items ordered before a certain date and delivered after it (the items being order-processed at that date). This means that both date predicates have a selectivity of roughly half, which in the classic merge-join processing case leads to a difference of factor two due to saved I/O in the clustered case, where these selections can be pushed down (i.e. lead to less data being touched). In this complex query, that consists of many (<200) primitive database operators ("MAL" statements) in MonetDB, we see that the absence of unnecessary joins in the RDFscan/RDFjoin case gives an order of magnitude improvement!

This is not all though, as for Q3 there is an opportunity of selection pushdown over relationship, powered by ZoneMap indexes: since we know that most orders are handled rather quickly and never more than 3 months in this dataset, the amount of selected items should be no more than 3 months worth of orders, hence 1/12 of the warehouse (which spans 6 years). As most orders are handled well within 3 months, the overall selectivity percentage turns out to be 5%.

From the point of view of a database query optimizer, it seems like a hopeless task to discover this, as all it has are two open-ended selection predicates on different properties. What is worse, it would lead to scans on

half of the data (half of the ORDERS: those before the date) and half of the LINEITEMS (those after the date). However, our clustering scheme combined with the ZoneMap indexing is in fact able to avoid scanning 95% of the data, because the ZoneMap technique is able to translate the l_shipdate into an upper-bound on LINEITEM subject OIDs, and hence into also a lower bound on ORDER subject OIDs. The orderdate clustering provides a lower bound on ORDER subject OIDs as well as on LINEITEM subject OIDs.

Hence, as a result, this query scans and processes 5% of data instead of 50% making it 10 times faster in the mergejoin case and 6 times faster in the RDFscan/RDFjoin case.Overall, in the cold experiments, our optimizations accelerate MonetDB HSP by a factor >40.

2.3 MonetDB RDF Research Roadmap

Emboldened by the promising initial experimental results, CWI will be working further on the topic of clustering in RDF. Additionally, rather than up-front indexing, we would like to move a system that adapts at runtime to an evolving workload, following a similar philosophy as in Database Cracking [52]. Ideas regarding such run-time data reorganization and indexing are elaborated in Section 2.3.2.

2.3.1 Advanced Characteristic Set RDF Clustering

The current research in characteristic set clustering is aiming at providing an extended clustering algorithm that reduces the amount of characteristic sets further and is able to derive schema information. Further, this is coupled with a multi-dimensional clustering algorithm.

We consider Characteristic Sets (CS) auto-detected object "classes"; and the properties are object "attributes". Just like in UML, these attributes can have multiplicities like: 0..1, 1..1 and 0.. In RDF terms, one can also see a CS as a RDFS Class definition (inclusive property definitions) that is derived from the data, but RDFS does not have support to specify property multiplicities (0..1, 1..1, 0..n).

The paper that introduced characteristics sets [21] describes a classification algorithm. We extend the initial algorithm in thee ways:

- grouping CS's by generalization
- support for typed properties. We do not only consider properties, but the type of those properties as well, to split characteristic sets.
- Detecting relationships (a la “foreign keys”) between CS's.

Generalization. In contrast to the original CS algorithm, we further group the initial CS's into a larger CS descriptions which has a superset of all attributes, keeping track of which attributes are 1, which are 0..1 and which are 0..n in metadata. In the original paper, if a subject would be missing a property, it would lead to a separate characteristic set. In our approach, it leads to the property becoming 0..1 (or 0..n) if there is enough support -- if there are enough subjects that miss that property.

Typed Properties. After defining the initial set of CS's we further analyze the type of the properties that have been grouped. This means that we look at the literal type of the property.

Relationships. In case of URI properties, we type them using CS membership. This means that we split characteristic sets such that their types become homogeneous. We only keep those CS's that are still frequent.

The above process also leads to a foreign key graph between CS's. That is, if a property of one CS always refers in the object field to another CS, this is a foreign key between these two CS's. Note that this foreign key graph in itself will tend to be useful for SPARQL users. Here, its main role is to guide RDF storage.

RDFjoin/RDFscan. The CS-wise storage of triple data can be exploited by an algorithm for SPARQL star query execution assembles (S,O1,..On) bindings for multiple properties P1,..Pn in one go. This algorithm would determine which CS’s overlap with all these properties (or interset, for those Pi that might be marked OPTIONAL). It would then scan those CS’data exploiting the fact that the property-object data is fully aligned. This algorithm is much faster than multiple classical merge-joins. Further, it would accept als Pj-Oj conditions where both Pj and Oj are bound, and consult the zone maps to see if all triples in certain ranges qualify, or if certain ranges can be excluded a-priori. Only for those triples that do not meet these criteria, it would need to evaluate the condition.

Multi-Dimensional Indexing. The advanced part of characteristic set clustering is the order of the S identifiers. S-identifiers are large integers (64-bits OIDs). The question that arises is: what criteria determine the S-OID order? This implies finding zero or more properties of interest for each CS. One starting possibility is to let this be triggered by user hints; but better would be to let this be determined through workload analysis automatically. The fact that there are potentially multiple such criteria brings CS clustering in the realm of multi-dimensional indexing.

For each characteristic set, a number of such **dimensions** can be used over path expressions. A path expression is a fixed-length sequence of properties, with the restriction that these must be a [0..1] or [1..1] attribute in the relevant characteristic sets. Note that using paths of length more than one, we are traversing foreign keys between characteristics sets.

A dimension can be thought of as a property, but more generally as a property discretized in a limited number of so-called "bins". For example, for date properties, we might bin these per year, quarter or month. Automatic binning schemes that basically create bins by creating an equi-height histogram (or estimating such a histogram from a sample) extend this to any kind of property. In the end, each bin is just identified with a number (an integer). Dimension creation should try to find frequency-balanced bin boundaries, such that when applied for clustering, there is roughly the same amount of data items corresponding to each bin. This then gives the dimension optimal splitting power for selection pushdown.

Each characteristic set can thus be divided in cells over a multi-dimensional space. We then store the subjects **inside** a cell in an arbitrary identifier order, but the bin numbers of the dimensions form the major bits of the S-OIDs (so dimensions determine a rough ordering on S). Taking TPC-H (RDF-H) as an example, we can create dimensions:

- nationkey
- orderdate

And cluster SUPPLIER and CUSTOMER (two CS's) on nationkey. We can cluster ORDERS (a CS) on orderdate; but also on customer-nation (i.e. following a CS relationship "has_customer" to CUSTOMER). We can cluster LINEITEM on order-date, customer-nation and supplier-nation. The fact that certain CS's share some common dimensions is called "co-clustering".

This co-clustering on dimensions can be exploited in queries for selection pushdown across multiple characteristic sets. For instance, a query that restricts SUPPLIER to a certain REGION, allows not only pushing down the selection into the scan for SUPPLIER, but also the scan for LINEITEM. Further benefits of the multidimensional co-clustering can be found in aggregation and join operations where the dimensions are determined by the aggregation/join key.

- If two CS's are joined and share common dimensions, a hash-join can be turned into a partitioned hash-join joining only matching groups. As such, the memory requirements for large joins can be reduced.
- For aggregations, it is possible to compute aggregate results for each group. Thus, one big aggregation that consumes a lot of memory can be turned into many subsequent small aggregations, also reducing peak memory usage [44].

There are some SPARQL-specific opportunities and challenges to this kind of multi-dimensional indexing:

- we might also want to treat the graph to which a triple belongs as a special kind of dimension, even though the G is not an explicit RDF property.
- it is natural to use RDFS subClassOf to represent the membership of a subject to a dimension hierarchy. Expanding all the inferred triples (if productX has rdfs:type mp3player, then it also has rdfs:type audio and also rdfs:type electronics) will lead to rdfs:type to be a 0..n attribute that cannot be used for multi-dimensional indexing. We could possible create a stratification of elements of a certain rdf:type that form a rdfs:subClassOf hierarchy with each other. Membership of one layer of this hierarchy could be made explicit in generated RDF triples; S? productLevel1 classX (productLevel1 is the invented name for the inferred level of the hierarchy and classX is the URI of the class). This would be a 0..1 attribute that could be used for multi-dimensional indexing.

2.3.2 Fully Self-Organizing RDF Storage

The thesis [52] convincingly argues for database systems adopting techniques and an architecture that allows it to continuously react to the data that users store in a database and to the way database users query it. As

such, each insert/delete and query **is treated as an advice** how to store the data in the future.

This approach to query processing allows taking the human out of the database design loop. This is valuable, not only because human time is valuable, but also because users and even DBAs generally find it difficult to explicitly elicit what the query workload is. This is on the one hand an issue of the difficulty in understanding the implications of large user-generated query log and a many statistics. In the case of RDF storage, this is worsened by the fact that there is no explicit schema. This makes it even difficult to capture access patterns in any formal way, as in principle the data is a pile of edges without obvious structure.

Automatic CS Re-Clustering. We have seen that despite the lack of schema in RDF we can devise techniques like Characteristic Sets multi-dimensional RDF clustering. Such an indexing schema, while very powerful, requires deep insight in which are the important dimensions in a workload. Given the general difficulty in understanding schema, data and workload, determining a good indexing scheme is a difficult talk for users.

Consequently, the best way forward is to learn these important access dimensions on-the-fly. The CS clustering already performs regular re-clustering to add newly inserted RDF triples into the proper representation that exploits the structure of the characteristic set (see Section 4.3). During such periodic re-clustering, the indexing dimensions of interest could be adjusted. As such, we need learning algorithms to detect important patterns in a workload and include these in the automatic RDF data storage scheme.

Run-time Partial Value Indexing. For lookup-workloads where locality my not be the prime interest (due to low data volume per query) but where full table scans to locate data items are to be avoided, the exhaustive indexing strategy works rather well. Still, creating a full index (=table order; and even multiple such orders) for all properties and all values is likely overkill; because real workloads tend to focus on certain properties only and certain object ranges within these.

To counter this, we could investigate partial indexing strategies that only cover part of the objects and part of the properties. The immediate question then becomes: which parts? In the vein of Database Cracking [52], such decisions would best be made in reaction to queries as they occur in the workload. Hence, this research topic intends to bring graph value cracking to realm of RDF.

Run-time Partial Path Indexing. Taking the previous idea forward and towards the topic of graph query processing, we have seen that graph traversals over multiple steps e.g. through a social network friends structure quickly explode (within 6 steps one reaches the entire population). Still many relevant database queries where steps to a graph are made in conjunction with filters may end up producing small final results. The problem in query processing, even in case on both ends of a traversal path there are strongly limiting filters is that in the middle of the path the size of intermediate result may explode.

Path indexing can help alleviate such problems; in path indexing the (source, destination) results of a complex path consisting of multiple steps and path filters is retained in a separate structure. The final representation can be considered a join index. Now, the path can be evaluated rather than by navigating step-by-step (join-by-join) through the edge table (triple table), by instead joining against this join index in one go. The problem in path indexing again is that it is not obvious a priori which paths are: (i) valuable for the future workload (ii) truly save effort: the result of the path expression should be much smaller than the sum of the intermediate results one incurs by navigationally evaluating the path to be truly valuable.

Further, we may not want to create full path indexes that contain the (source, destination) of **all** nodes in the graph. If the workload only focuses on a small subset of relevant nodes, again we would like to create **partial** path indexes. The further question (iii) is thus for which subsets of source/destination nodes such path indexes should be kept.

In order to deal with issue (i), (ii) and (iii) it is obvious that again an adaptive approach is required that reacts continuously to the workload. For instance, it could create partial value and path indexes as a side-effect of query processing. Thus, while a first invocation would incur significant computation, certain high-value the intermediates produced could be added to a partial index much in the vein of Recycling [53], to accelerate future queries.

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 of 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/semantictech/ 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).

[40] TSIALIAMANIS, P., SIDIROURGOS, L., FUNDULAKI, I., CHRISTOPHIDES, V., AND BONCZ, P. Heuristic-based Query Optimization for SPARQL. In EDBT (2012).

[41] KAOUDI, Z. Distributed RDF Query Processing and Reasoning in Peer-to-Peer Networks. PhD thesis, Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Greece (2011).

[42] BRODT, A., SCHILLER, O., MITSCHANG, B. Efficient Resource Attribute Retrieval in RDF Triple Stores. CIKM (2011).

[43] ANTONELLI, M. A SPARQL front-end for MonetDB. MSc. Thesis Universita degli studi ` "Roma Tre" (2008).

[44] BAUMANN, S., BONCZ, P., SATTLER, K-U.– Bitwise Dimensional Co-Clustering in Column Stores. Unpublished document (www.cwi.nl/~boncz/main.pdf).

[46] SUCHANEK F., KASNECI, G., WEIKUM, G. YAGO - A Core of Semantic Knowledge. WWW 2007

[47] ERLING, O. Virtuoso, a Hybrid RDBMS/Graph Column Store. IEEE DEBULL. March 2012.

[48] ZUKOWSKI, M. BONCZ, P. Vectorwise: Beyond Column Stores. IEEE DEBULL. March 2012.

[49] LAMB, A., FULLER, M., VARADARAJAN, R., TRAN, N., VANDIVER, B., DOSHI, L., BEAR, C. The Vertica Analytical Database: C-Store 7 Years Later. VLDB (2012).

[50] PADMANABHAN, S., BHATTACARJEE, B., MALKEMUS, T., CRANSTON, L., HURAS, M., Multi-Dimensional Clustering: A New Data Layout Scheme in DB2. SIGMOD (2003).

[51] O'NEIL, P., CHENG, E., GAWLICK, D., O'NEIL, E. The Log-Structured Merge Tree (LSM-Tree). Acta Informatica 33(4):351-385. (1996)

[52] IDREOS, S. Database Cracking: Towards Auto-tuning Database Kernels. University of Amsterdam PhD Thesis (2011). ACM SIGMOD Jim Gray Disstertation Award 2011.

[53] IVANOVA, M., KERSTEN, M. NES, N., GINCALVES, R. An architecture for recycling intermediates in a column store. SIGMOD (2009).

[54] AGRAWAL, R., MANNILA, H., SRIKANT, R., TOIVONEN, H., INKERI VERKAMO, A.: Fast Discovery of Association Rules. Advances in Knowledge Discovery and Data Mining 1996:307-328

[55] RAVINDRA, P. DESHPANDE, V., ANYANWU, K., Towards scalable RDF graph analytics on MapReduce. Workshop on Massive Data Analytics on the Cloud. (2010).

3. Appendix

The appendix contains the SP2Bench and YAGO queries that were used for the experiments conducted with the heuristic-based SPARQL planner HSP

3.1 SP2Bench Queries

3.1.1 Query SP1

```
SELECT ?yr ?journal
WHERE {
        ?journal rdf:type bench:Journal .
        ?journal dc:title "Journal 1 (1940)" .
        ?journal dcterms:issued ?yr .
}
```

3.1.2 Query SP2a

```
SELECT ?inproc
WHERE {
        ?inproc rdf:type bench:Inproceedings .
        ?inproc dc:creator ?author .
        ?inproc bench:booktitle ?booktitle .
        ?inproc dc:title ?title .
        ?inproc dcterms:partOf ?proc .
        ?inproc rdfs:seeAlso ?ee .
        ?inproc swrc:pages ?page .
        ?inproc foaf:homepage ?url .
        ?inproc dcterms:issued ?yr .
        ?inproc bench:abstract ?abstract .
}
```

3.1.3 Query SP2b

```
SELECT ?inproc
WHERE {
        ?inproc rdf:type bench:Inproceedings .
        ?inproc dc:creator ?author .
        ?inproc bench:booktitle ?booktitle .
        ?inproc dc:title ?title .
        ?inproc dcterms:partOf ?proc .
        ?inproc rdfs:seeAlso ?ee .
        ?inproc swrc:pages ?page .
        ?inproc foaf:homepage ?url .
        ?inproc dcterms:issued ?yr .
}
```

3.1.4 Query SP3a

```
SELECT ?article
WHERE {
        ?article rdf:type bench:Article .
        ?article ?property ?value
        FILTER (?property=swrc:pages)
}
```

3.1.5 Query SP3a_2

```
SELECT ?article
WHERE  {
        ?article rdf:type bench:Article .
        ?article swrc:pages ?value
}
```

3.1.6 Query SP3b

```
SELECT ?article
WHERE {
        ?article rdf:type bench:Article .
        ?article ?property ?value
        FILTER (?property=swrc:month)
}
```

3.1.7 Query SP3b_2

```
SELECT ?article
WHERE  {
        ?article rdf:type bench:Article .
        ?article swrc:month ?value
}
```

3.1.8 Query SP3c

```
SELECT ?article
WHERE {
        ?article rdf:type bench:Article .
        ?article ?property ?value
        FILTER (?property=swrc:isbn)
}
```

3.1.9 Query SP3c_2

```
SELECT ?article
WHERE {
        ?article rdf:type bench:Article .
        ?article swrc:isbn ?value
}
```

3.1.10 Query SP4a

```
SELECT ?person ?name
WHERE  {
        ?article rdf:type bench:Article .
        ?article dc:creator ?person .
        ?inproc rdf:type bench:Inproceedings .
        ?inproc dc:creator ?person2 .
        ?person foaf:name ?name .
        ?person2 foaf:name ?name2
        FILTER (?name=?name2)
}
```

3.1.11 Query SP4b

```
SELECT ?person ?name
WHERE  {
        ?article rdf:type bench:Article .
        ?article dc:creator ?person .
```

```
        ?inproc rdf:type bench:Inproceedings .
        ?inproc dc:creator ?person .
        ?person foaf:name ?name
}
```

3.1.12 Query SP5

```
SELECT ?s ?p
WHERE {
        ?s ?p person:Paul_Erdoes
}
```

3.1.13 Query SP6

```
SELECT ?ee
WHERE {
        ?publication rdfs:seeAlso ?ee
}
```

3.2 A. YAGO Queries

3.2.1 Query Y1

```
SELECT ?GivenName ?FamilyName
WHERE  {
        ?p yago:hasGivenName ?GivenName .
        ?p yago:hasFamilyName ?FamilyName .
        ?p rdf:type yago:wordnet_scientist_110560637 .
        ?p y:bornIn ?city .
        ?city yago:locatedIn yago:Switzerland .
        ?p yago:hasAcademicAdvisor ?a .
        ?a y:bornIn ?city2 .
        ?city2 yago:locatedIn yago:Germany .
}
```

3.2.2 Query Y2

```
SELECT ?a
WHERE {
        ?a type  wordnet_actor_109765278 .
        ?a livesIn ?city .
        ?a actedIn ?m1 .
        ?m1 type wordnet_movie_106613686.
        ?a directed ?m2 .
        ?m2 type wordnet_movie_106613686.
}
```

3.2.3 Query Y3

```
SELECT ?p
WHERE {
        ?p ?ss ?c1 .
        ?p ?dd ?c2 .
        ?c1 type wordnet_village_108672738 .
        ?c1 locatedIn ?X .
        ?c2 type wordnet_site_108651247.
        ?c2 locatedIn ?Y .
}
```

3.2.4 Query Y4

```
SELECT ?p1 ?predicate ?p2
WHERE {
        ?p1 ?u1 ?city1.
        ?city1 label Paris .
        ?p1 ?predicate ?p2.
        ?p2 ?u2 ?city2.
        ?city2 label Hong Kong .
}
```

3.3 Blueprint of Future MonetDB RDF support

In the following we outline the new architectural direction for RDF support in MonetDB. As a word of warning, this section will progressively get more detailed and MonetDB-specific.

3.3.1 Critique of the Original MonetDB RDF support

The original MonetDB RDF design lacked on four fronts, novelty, compactness, updates and advanced SPARQL features.

- **Novelty.** The chosen approach of exhaustive indexing has been demonstrated many times. As described above, all this indexing still does not lead to real locality that can match properly designed relational data warehousing solutions.

  On the other hand, the work on characteristics sets shows perspectives to get all the indexing and locality benefits that are within reach for relational practitioners. Bringing this to the world of RDF would truly enhance the state-of-the-art in RDF stores.

- **Compactness.** Data storage size in RDF systems is of crucial importance, due to this lower locality. RDF stores typically only perform well if the hot-set of the database fits in the main memory; because random access to disk-resident data is very slow. Random access to RAM is tolerable (although not efficient either as there will be many TLB and CPU cache misses).

  The database hot-set usually means the SPO/POS/...etc../OPS integer tables (or B-trees). Apart from these integers, there are mappings from integer to URIs and values. These mapping tables are typically less hot; for OLAP workloads that count things they often are not even queried; and for OLTP queries the access is very sparse such that a few I/Os are tolerable.

  Thus, replicating S,P,O information many times has the drawback that it increases data volume. Another critical feature in RDF stores is therefore compression: an ordered table like PSO is highly compressible.

  Both replication and compression were a drawback of the original MonetDB RDF plan: the shredder creates all 6 orders of PSO. Further, it does not apply any compression, storing the identifies as (8-byte) OID numbers; thus the original design needs 150 bytes per triple; including string literals and the URI dictionary. We note that the URI mapping does apply compression, and is relatively efficient.

- **Updates.** Updates in SPARQL consist of triple deletes and inserts. The original MonetDB SPARQL with its replicated ordered tables made updates quite expensive due to this replication. Also, it is not really possible to update ordered tables without fully copying them, in the MonetDB data structures. All in all that design was quite update unfriendly.

  One might argue whether SPARQL workloads really need updates. There seem to be quite a few that do not. So, this is not a must-have feature. Still a database system without updates is a bit like a car without steering wheel; and many applications do need updates.

- **Useful Features.** Finally, the original MonetDB RDF design did not have support for:
    - typed literals: even falling short of full compliance with the SPARQL type system, any useful system should go beyond just supporting string literals.
    - graphs: the notion that triples belong to a set (a graph) is by now assumed standard by SPARQL users
    - inference: support for rdfs:subClassOf, owl:equivalentClass, owl:equivalentProperty, rdfs:subPropertyOf and owl:sameAs in SPARQL evaluation (i.e. the ability to use property synonyms, rdfs:type synonyms and subject synonyms).

3.3.2 Adaptive RDF Storage: basic SPOG + clustered PSOG

Our proposed storage scheme uses two representations, namely

- a **basic representation** and
- a **clustered representation**.

All triples are stored in one of these two representations. Typically, a dataset is first loaded in basic representation, and then (partially) moved to clustered representation following reorganization.

In the spirit of MonetDB we target analytical RDF applications. For such purposes, the PSOG order is best as it resembles the column-store approach. However, for noisy and unclassified data we initially prefer SPOG storage.

Basic storage: SpOg. Basic triple storage consists of SPOG; where S is of type OID (64-bits), O is of type lng (also 64-bits) and P and G are int-s (32-bit). We write SpOg to suggest that ‘p’ and ‘g’are smaller integers than ‘S’and ‘O’. The 32-bits integers used for ‘p’ and ‘g’ imply that MonetDB SPARQL has a restriction on max 2G properties and graphs.

Even though we favour PSOG due to its "column store" spirit, the reason to choose SPOG for basic storage is that it will end up with the irregular part of an RDF dataset, where properties will be infrequent. In case of infrequent properties, for S-access the merge-joins on S-streams for each P would drown due to way to many different (infrequent) P's in the PSOG scheme. SPOG is of course fine for S-access and for P-access we can rely on the MonetDB hash index on the P column.

SpOg order is in fact just Sp order: we will not enforce sub-order on Og.

The representation of the objects represents every O as a 64-bits integer (MonetDB data type: lng). This is done such that whatever the expression/subquery and intermediate result, we can represent values in a single and efficiently processable column.

The highest 2 bits are exploited as follows:

- 00 dateTime (lowest 62 bits numeric)
- 01 numeric (lower 58 bits is the number, lowest 4 bits is lexical type)
- 10 URI (an OID that points into the URI dictionary).
- 1000000000000...0000000000 is a special value that indicates a deleted triple (should be ignored)
- 11 string (an OID that is must me looked up in another table) bits 2..16 (14 bits) are an index to a string mapping table.

During string shredding, we will not store all strings in a single mapping table; rather value-partition the strings in multiple buckets (at most 8192 buckets in the shredder). During query processing, intermediate expressions may create additional new string values, which will be added in new temporary buckets. The even buckets are the shredded buckets, and the odd buckets are the temporary buckets used during query processing. This is a way to (i) provide isolation yet benefit from global string mapping tables (ii) enhance string lookup using the bucket approach (the matching bucket allows to pre-filter a range restriction on string

using a range restriction on the object OID-s).

XML is as yet unknown and will just be treated as xsd:string. Strings with a language are supported by prefixing them with a <lang>@tag (normal strings start with @).

For the numeric types, the lower four bits are exploited as follows:

- 0 xsd:double
- 1 xsd:decimal
- 2 xsd:integer
- 3 xsd:boolean
- 4 xsd:nonPositiveInteger
- 5 xsd:negativeInteger
- 6 xsd:long
- 7 xsd:int
- 8 xsd:short
- 9 xsd:byte
- 10 xsd:nonNegativeInteger
- 11 xsd:unsignedLong
- 12 xsd:unsignedInt
- 13 xsd:unsignedShort
- 14 xsd:unsignedByte
- 15 xsd:positiveInteger

Note that this mapping respects integer range order. Both equi- and range-selections map to 64-bits integer ranges. The integers are stored as multiples of 100000 (5 digits) so that decimals and integers are comparable. xsd:double is stored identically to xsd:decimal (rounded to nearest on shred).

Given that we use two bits to encode a lng to represent some numeric, and then 4 bits for its lexical type; there are 58 bits left for the actual value. All in all this means that we have 58-bits signed integers, representing decimals with 5 places behind the comma. So the biggest decimal is 1441151880758.55871 and the smallest is -1441151880758.55872

xsd:float will be parsed and can be cast to, but all such information is stored and treated as xsd:double

In all, the typical storage cost of the basic scheme is 24 bytes per triple; plus the string size; not counting the URI mapping tables.

Handling Updates. Updating sorted tables in MonetDB is not really possible, as data is stored in arrays and not in B-trees. Therefore we propose the implementation of a module that stores data in multiple tables, that each differ from each by a large factor (X>>2) in size. As such, there will be logX(N) tables T_i, each with intended size $(c.X)^i$ tuples. Updates go to the smallest table. If a table gets to be twice its intended size, it is merged with the next biggest table, all tables smaller than it move up one place and a new smallest empty table is created. This approach is called **Log Structured Merge Tables**, a minor variant of Log Structured Merge Trees [51].

As the SPOG table will not be read by normal MAL commands, but by our RDFscan command instead, the fact that there are now multiple tables to worry about can be hidden.

A final idea for handling updates is targeted at the fact that in our scheme the basic storage in SPOG will

have to deal with the long tail of irregular and not often used properties. The more common regular cases will migrate to the structured storage scheme.

Clustered Storage: |p|*sOg. We implement a reorganization operator that runs the characteristic set algorithm on the full dataset. Subjects qualify as members of a CS if the subject possesses all "required" (1..*) properties. It may qualify for multiple CS's in which case it is classified to just one (with most properties). Its other triples go into basic storage then.

The metadata that we store keeps track of all detected characteristic sets, their attributes and their attribute multiplicities (0..1, 1..1, 1..n etc). We also store statistics such as the CS cardinalities that can be useful for query optimization.

The clustered storage most resembles conceptually PSOG, but split for each property in a separate table, such that we only store sOg (P constant). We again use the non-capital letters of 's' and 'g' in sOg to indicate that these columns are highly compressible. A sOg table itself may also be split into multiple partitions i.e. a Log Structured Merge Table; but typically there is a limited number of them (this is done to ease updates, because we must respect S order). You can also see this clustered storage for CS's as an S-table with one column for each P ("property tables" in SPARQL jargon), though of course fully automatic.

We will not enforce OG sub-order in these tables. In detail, these sOg storage of data belonging to CS's works as follows:

- s: we divide the S-es in runs that are of two types: (i) dense or (ii) ascending. We only store an OID with the base of each run. The highest bit contains whether it is dense or ascending. The next 7 bits contain a run length. Dense sequences are (base,base+1,..,base+size-1). Ascending sequences are created by using a second column of bytes that contains deltas. Thus, we can handle gaps of up to 255 (a large gap must use a new base, hence ends the run).
- O: just like in the basic storage, we store everything as a 64-bits integer (lng). However, we employ compression by storing these as an int column with the lower 32 bits and a RLE representation of the high bits in a combination of (int:value,byte:count). This exploits the fact that the upper 32-bits bits are often equal (or zero) and leads to more compact storage.
- g: similarly, the graph ID is also RLE stored as (int,byte), exploiting the fact that graphs will often be repeating.

In all, the typical storage cost of the clustered scheme is just over 4 bytes per triple (almost all spent in O); plus the string size; not counting the URI dictionary.

CS reorganization. When the primary basic SPOG table gets too big, we run the characteristic set algorithm and move triples into clustered storage. This will lead to URI renumbering; which is a headache. One simple strategy to limit impact is to never renumber a subject that is also used as a property - so we never need to maintain property columns (or metadata). It limits the work to objects.

Object renumbering does have to occur, both in the basic storage as well as in the clustered storage.

Simple Inference Support. For inferencing, the easy target is to exploit the classical trick of expand-on-load:

- rdfs:subClassOf/rdfs:subPropertyOf: generate extra triples implied by the subclass/subproperty relationship. Store these in a special graph that contains the derived triples for the original graph.
- owl:sameAs/owl:equivalentClass: choose one of the URIs as the primary one, and add the sameAs to a translation table that is used to map incoming URI literals in queries to the primary one.

These ideas are not truly novel, but may be enough in the short term.

3.3.3 Proposed MAL Extensions

We propose the addition of the following operations:

```
RDFscan(P1..Pn,D1[l1-h1]..Dm[lm-hm],G)
```

- generate all triple (S,O1,..On) bindings for all properties Pj
- we push down selections li<=Di<hi on dimensions Di. A Di is a fixed length property path, given as a string.
- hi/li are literal values given as a string (e.g. "42^xsd:integer"). G and Pj are URI strings.
- The result is a set of BATs. For each Pj we also pass whether it is obligatory match or not.
- G is a restriction on the graphs to be queried (or *)

```
RDFjoin(P1..Pn,D1[l1-h1]..Dm[lm-hm],G,S)
```

- S is a restriction on the subjects of interest (such that this becomes a join)

```
RDFselect(P1..Pn,D1[l1-h1]..Dm[lm-hm],G,P,O)
```

- we select on one property P for matching O-values given in parameter O

```
RDFjoinselect(P1..Pn,D1[l1-h1]..Dm[lm-hm],G,S,P,O)
```

- a combination of RDFjoin (S input restriction) and RDFselect (O output restriction)

Given that we have two representations of our datasets, we need to provide two different implementations for these.

- for the basic storage it consists of access to the SPOG for each named P, where the selection is pushed down -- only possible for paths D_j of length 1 (a range-restriction on a being scanned -- it may be added to the scan if D_j not in P_i).
- for the clustered storage the D_j are translated to dimensions and thus cell ranges. These in turn lead to restriction on the S ranges, which are intersected. We also use the obligatory P_i to narrow down the selected set of CS's; which also leads to S-ranges. These S-ranges are intersected with all the value ranges. Similar to basic storage, the result is again formed by a multi-merge-join on all P's. This merge-join hides all the decompression tasks.

Note that the result of the operations is always the union of both cases (basic and clustered).

We will also need some functions to convert from our integer representation of literals to real MonetDB literals and back.

In case of string-mappings, we will use a system with multiple string lookup tables (given by the major 2..X bits) such that we can add query-local mapping tables for newly created bits. The string-fetch fetches the strings depending on the mapping table bits from the right bats.