LDBC

Cooperative Project

FP7 – 317548

D2.2.1 Analysis and classification of choke points

Coordinator: Andrey Gubichev, Thomas Neumann

1st Quality Reviewer: name of first reviewer
2nd Quality Reviewer: name of second reviewer

Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	M6
Actual delivery date:	M6
Version:	0.1
Total number of pages:	27
Keywords:	linked data, query optimization

Abstract

This document surveys the existing approaches to store, index and query RDF data. While providing an overview of diverse techniques for RDF data storing, we concentrate on triple stores for query processing and query optimization issues. We describe the state-of-the-art solutions for two classical database management problems of join ordering and selectivity esimtimations in the context of RDF systems, and then analyze their drawbacks. Namely, we provide a detailed descriptions of technical challenges (coined choke points) that any system must cope with in order to efficiently execute complex SPARQL queries on large real-world RDF datasets. These choke points will later be used in the SPARQL benchmark design. Although illustrated with RDF/SPARQL examples, the classification of choke points is general, and applies to Graph DBs and declarative graph query languages.

Executive Summary

This report surveys the state-of-the-art of RDF querying in the centralized environment and describes the technical challenges that RDF systems have to cope with in order to efficiently execute SPARQL queries over large real-world datasets.

In the first part, we describe the approaches towards storing, indexing and querying RDF datasets. We start with reviewing different architectures of RDF storage systems, where the two important classes are relational-based approaches and graph-based approaches. The former one (and especially triple stores) have become the de-facto standard in storing RDF data. Furthermore, we review the approaches towards solving the classical datavase problems of join ordering and cardinality estimations in the context of RDF databases. We describe the RDF-specific statistics for cardinality estimations and present the RDF-tailored physical operators (joins and path traversal).

In the second part we analyze the technical challenges (*choke points*) that the query optimizer has to solve for getting the optimal plan for the query. We also provide illustrations of these challenges by SPARQL queries over a real-world large dataset. Specifically, the following challenges are described in the report:

- efficiently finding the optimal join ordering for star-shaped queries
- dealing with large search space for the join ordering problem
- considering different types of join trees (bushy trees) as candidates for the optimal plan
- handling the OPTIONAL clause in join ordering
- correctly esimating the cardinalities of star-shaped subqueries
- accurately esimating the cardinalities of subqueries in large queries
- choosing the right type of join operator depending on cardinality estimations
- constructing the path traversal operator depending on cardinality estimations

These problems will serve as a foundation for our SPARQL query benchmark.

We note that, although we focus on SPARQL and RDF in this deliverable, the results are applicable for Graph DBs and any graph declarative query language, e.g. Cypher. We discuss it in more detail in Section 2.3.5.

Document Information

IST Project Number	FP7 – 317548	Acronym	LDBC
Full Title	LDBC		
Project URL	http://www.ldbc.eu/		
Document URL	provide the URI for the document		
EU Project Officer	Carola Carstens		

Deliverable	Number	D2.2.1	Title	Analysis and classification of choke points
Work Package	Number	WP2	Title	Query Processing

Date of Delivery	Contractual	M6	Actual	M6
Status	version 0.1		final □	
Nature	Report (R) ☒ Prototype (P) □ Demonstrator (D) □ Other (O) □			
Dissemination Level	Public (PU) ☒ Restricted to group (RE) □ Restricted to programme (PP) □ Consortium (CO) □			

Authors (Partner)	Andrey Gubichev (TUM), Thomas Neumann (TUM)			
Responsible Author	Name	Andrey Gubichev	E-mail	gubichev@in.tum.de
	Partner	TUM	Phone	+49

Abstract (for dissemination)	This document surveys the existing approaches to store, index and query RDF data. While providing an overview of diverse techniques for RDF data storing, we concentrate on triple stores for query processing and query optimization issues. We describe the state-of-the-art solutions for two classical database management problems of join ordering and selectivity esimtimations in the context of RDF systems, and then analyze their drawbacks. Namely, we provide a detailed descriptions of technical challenges (coined choke points) that any system must cope with in order to efficiently execute complex SPARQL queries on large real-world RDF datasets. These choke points will later be used in the SPARQL benchmark design. Although illustrated with RDF/SPARQL examples, the classification of choke points is general, and applies to Graph DBs and declarative graph query languages.
Keywords	linked data, query optimization

Version Log			
Issue Date	Rev. No.	Author	Change
14/02/2013	0.1	Andrey Gubichev, Thomas Neumann	First version

Table of Contents

1 Introduction

Last years have seen an impressive growth of linked data on the web. According to the latest diagram of the Linked Open Data cloud, there are 31,634,213,770 (over 31 billion) triples published online. Large individual datasets contain hundreds of millions of triples and span domains like biomedical research (Uniprot, PubMed), knowledge bases (DBpedia, YAGO), open government data (Data.gov), entertainment (MusicBrainz) and others. These numbers clearly indicate the need of efficient indexing of RDF datasets and scalable SPARQL query processing, which is the subject of this chapter. Managing large-scale RDF datasets impose the following technical difficulties for storage, indexing and querying:

- How to index the diverse (and potentially dynamic) RDF datasets? The absence of a global schema and diversity of resource names (string and IRIs) make this task very challenging. While most of the commercial engines use triple stores as physical storage for RDF data, such storage requires efficient indexing scheme to provide fast lookups and support join operators.
- How to find the optimal execution plan for complex SPARQL queries? Since the data model essentially consists of triples (as opposed to records in relational model), the extraction of different attributes will boil down to large number of joins. Both scalable join operators backed by suitable indexes and efficient query optimization strategies are needed.
- How to provide the query optimizer with accurate estimations of result sizes? This typically calls for RDF-specific data structures that would handle the selectivity and cardinality estimations for schema-less RDF data, taking into account typical SPARQL queries like long join chains and large join stars over many-to-many relationships.
- How to efficiently execute the physical operators like joins and path traversals? Even after finding the optimal execution plan, a lot of heavy-weight self-joins need to be run against the single table. Path traversals also become challenging on large datasets since they "touch" significant portions of the data.

In the first part of this report we will review the state-of-the-art approaches towards these problems. We start with describing different approaches to RDF storing and indexing, then turn our attention to query processing and optimization, focusing on join ordering and cardinality estimations, and conclude with surveying join and path traversal processing in RDF databases.

We note that although the research community has also looked into handling large volumes of RDF data in the distributed setting, in this chapter we concentrate solely on storing and querying RDF datasets in the centralized environment (i.e., we consider an RDF graph located on a single machine).

In the second part of the report we analyze specific technical challenges that the cost-based query optimizer must deal with in order to execute complex SPARQL queries over large real-world datasets. These are the choke points that would serve as a foundation for our SPARQL query benchmark. More specifically, we will provide multiple examples of the following two classes of choke points:

- Join ordering challenges, more specifically the trade-offs between the time spent to find the best execution plan and the quality of the output plan. Finding an exact solution sometimes is prohibitedly expensive, but at the same time naive heuristics frequently miss good plans
- Cardinality estimation challenges. Naturally, the exact result size of the query is known only after its execution, so only heuristics-based guesses are available to the optimizer. Accurate estimation is usually hindered by specific properties of the data like irregularities and implicit correlations.

We also point out how these SPARQL query optimizer's challenges correspond to a Graph DB's optimizer challenges.

2 Core Part

2.1 Storing and indexing of RDF data

Conceptually we can distinguish between two different approaches towards storing RDF data. One, the *relational-based* approach, views an RDF dataset as a set of triples (S, P, O), which can be considered as a special case of relational data model. This allows applying techniques of storing and indexing developed for the relational databases to RDF with their potential specialization. Such an approach also leverages the similarity between SPARQL and SQL. The second, *graph-based* approach, suggests looking at an RDF dataset as a graph where subjects and objects correspond to nodes connected with predicates-edges. This line of work focuses on developing native graph structure indexes for RDF.

2.1.1 Relational-based approach

Most publicly accessible RDF systems and research prototypes employ the mapping of RDF datasets into relational tables. In principle, one can do so in one of the following three ways:

Triple stores put all triples into a giant triple table with the *subject*, *predicate*, *object* attributes. In some systems, an additional *graph* column is included to record the named graph that contains the triple [29, 24, 4]

Property tables (also coined *vertical partitioning* approach [1]), where triples are grouped by predicate name, and all triples with the same predicate are stored in a separate table. This extreme approach can be naturally implemented within a column store [25].

Cluster-property tables Triples are grouped into classes based on co-occurrence of sets of predicates in the dataset. Then each triple is stored in the table that corresponds to its class, with attributes being named after the predicates from the class [26, 30] The classes are typically determined by a clustering algorithm or by an application expert.

Naturally, in order to support complex query processing, efficient indexing schemes have been introduced for each representation. In the rest of this section we will consider these three approaches in more detail. Since the majority of existing systems and prototypes build on the triple store model, we will first look into indexing of such stores with more details, and later will survey the property table and cluster-property table approaches.

Triple store indexing

One of the first proposals for improving the query performance was to use an *unclustered index* on several combinations of S, P, O attributes of the triples table. Namely, the following five distinct combinations of B+-tree indexes were compared in [16]: (a) separate indexes on S, P, O, (b) joint index on S and P, (c) joint index on S and P and a separate index on O, (d) joint index on P and O, (e) a single joint index on S,P,O. Experiments on simple lookup queries and queries with few joins showed that the first index selection yields the best performance.

While the unclustered indexes merely store pointers into there triple table, it seems unavoidable that for complex queries *clustered indexes* that store multiple copies of the sorted triple table will perform significantly better, as having the data sorted in different orders allows for using merge joins for a large class of queries.

In order to keep space requirements reasonable for excessive indexing, systems typically use the dictionary compression. Namely, each IRI and literal in the dataset is mapped to a unique ID, and data is stored as triples of integers, not strings. There are two ways of assigning such IDs. First, the incremental ID approach can be used, where every new data item gets an increased internal ID [24, 3]. The string to ID and the ID to string mappings themselves are stored as the B^+-trees. Second, an ID for a string can be generated by a hash function [9]. Then, only the ID to string mapping needs to be stored. In this scheme hash collisions may lead to rejection

of the triple insertion. Since the same string may sometimes be used as both IRI and literal, an additional flag is stored to distinguish between the two.

Hexastore [29] employs six different clustered B+-tree indexes on all possible permutations of S, P and O: SPO, SOP, PSO, POS, OSP, OPS. For example, in the OPS ordering index, every distinct object is associated with the vector of predicates that occur in one triple with that object, and every predicate in turn is appended with the list of subjects. Weiss et al [29] note that in this architecture, the two indexes OPS and POS share the same list of subjects, so only a single copy of that list needs to be stored. Similar observation holds for two other pair of indexes. Hence, the theoretical upper bound on space consumption is five times the size of the triple table, and much lower in practice. Such an indexing scheme guarantees that we can always answer every triple pattern with variables in any position with a single index scan.

RDF-3X [24] follows a similar aggressive indexing strategy and stores all 6 permutations of triples in B^+-tree indexes. In addition, all subsets of (S, P, O) are indexed in all possible orders, resulting in 9 additional indexes. In these indexes, the parts of a triple are mapped to the cardinalities of their occurrence in the dataset. For instance, the aggregated index on PO matches every pair of predicate and object to the number of subjects with this predicate and object, i.e. every (p', o') maps to the $|\{s|(s, p', o') \in G\}|$, where G denotes the RDF graph. These aggregated indexes are later used to estimate cardinalities of partial results during query optimization and to answer queries where full scans are not needed. The following query computes all bindings of *?s* that are connected with the specified predicate *p* to any object, the actual value of the latter is not relevant. In this case the aggregated index PS is used:

```
select ?s
where {?s p ?o}
```

The semantics of SPARQL forces us to produce the right number of duplicates for the *?s* binding. Therefore, each aggregated index also has the count field, namely the number of the occurrences of the corresponding pair.

In order to make space requirements reasonable for such an aggressive indexing, RDF-3X employs the delta compression of triples in the B^+-tree leaves. Since in the ordered triples table two neighboring triples are likely to share the same prefix, instead of storing full triples only the differences between triples (*deltas*) are stored. Every leaf is compressed individually, therefore the compressed index is just a regular B^+-tree with the special leaf encoding. Neumann and Weikum [24] note that while compressing larger chunks of data would lead to a better compression rate, the page-wise compression allows for faster seek and update operations in the corresponding B^+-tree.

A less aggressive scheme is used by the TripleT technique [6]. It reduces redundancy by keeping just one B^+-tree containing every literal/IRI resource. Each such resource is mapped to a bucket (*payload*) of triples where it occurs. The payload for a resource is kept as three sorted groups corresponding to the position where this resource occurs in the triple. The choices of sort ordering within the payload are based on the assumption that subjects are more selective than objects which in turn are more selective than predicates. The advantage of such approach is increased data locality, since all the triples related to one resource are located together. This naturally comes at a cost of limiting the options available for query optimizer since not all orderings are available.

Virtuoso [4, 5] employs a partial indexing scheme, where only two full indexes on quadruples (PSOG, POGS) are stored. In addition, three partial indexes SP, OP and GS are kept. There, the index POGS is a bitmap for lookups on object value.

Taking this bitmap idea to an extreme, BitMat [2] stores triples into the large bit-cube, where each direction corresponds to S, P and O. The slices of the cube along these directions (i.e., bit matrices) are stored as row-wise gap-compressed data. Such slices essentially simulate all different permutations of S,P,O, except the two combinations: Atre et al [2] claim that the O-P projection in the direction S and the S-P projection in the direction O are rarely used and therefore are not stored, but can be reconstructed on the fly if needed. Similar considerations lead to using the position of the literal/IRI along the dimensions as the integer ID: although such encoding results in potential overlap between Subjects and Predicates IDs (and Predicates and Objects IDs as well), the corresponding joins S-P and O-P are extremely rare in the workload and are ignored by the system.

In addition to B^+-trees and bitmaps, some systems also use hash-based indexes. For example, the YARS2 system [10] suggests indexing the quadruples (GSPO) and all their embedded pairs and single values in 6 differ-

ent indexes (B^+-tree and hash). However the system only considers one ordering of tuples and therefore aims at supporting simple lookup queries rather than queries with multiple joins. Alternatively, Harth et al [10] consider using an in-memory sparse index instead of B^+-trees, in which each entry refers to the first record of a sorted disk-resident block. Lookup of an entry is then performed with binary search in memory followed by reading the candidate block from disk.

Property and cluster-property tables

Early work has investigated storing RDF data in multiple tables that are defined by one or several predicates [1, 15, 26, 30]

An extreme approach to such multiple table setup leads to using a separate table for each predicate [1]. Each table contains tuples of form (S, O). Subjects connected to several object via the same predicate result in multiple rows with the same subject and different objects. The subject column is sorted, thus allowing for fast merge joins on subjects. In addition, objects can be additionally indexed via unclustered B^+-trees or by replicating each table and sorting the clone table by object.

Jena [30] supports cluster-property tables for single-valued properties, i.e. for the set of properties $p_1, \ldots, p_l$ such that there exists at most one s and o with the triple (s, p_i, o) in the dataset. A single-valued property table has a schema of a subject (a key) and a group of frequently co-occurring predicates and stores object values for the properties with the cardinality of one. Conceptually each row describes an entity with the attributes specified by the group of predicates. Multiple-valued properties (e.g., different books of the same author etc) are stored in the vertical tables similar to [1], with every (S,O)-pair resulting in a tuple. Remaining triples that belong to rare classes, or whose not belonging to any class, are stored in the triple table.

In order to automatically define the set of properties for the cluster-property table, Sintek et al [26] suggest finding signatures of each resource x as $\{p|\exists o, (s, p, o) \in G\}$. Then, the table is created for every signature set. Since this strategy typically leads to creation of many small tables for diverse datasets, the authors suggest several strategies of merging these small tables into larger ones. In addition to the signature technique, the association rule mining method can help finding the predicates that frequently occur with the same subject [15].

Clustering of triples into tables usually assumes significant degree of regularity in the dataset. It is observed in [30] that such an approach requires an application expert to construct a schema in advance, thus significantly limiting the flexibility compared to triple stores. Alternatively, rather complex clustering algorithms need to be employed by the system [15]. Moreover, the schema has to be changed once a new property is introduced in the dataset (with, say, an update of a triple). This differs from the classical relational database scenario, where the database schema is static.

Besides, a major drawback of both property and cluster-property approaches is that it does not support queries with unbounded properties [1, 30]. So, if the query does not restrict a property of an entity, or the value of some property is runtime bound, all tables need to be scanned, and the results have to be combined with unions or joins. This is obviously far from optimal for large and diverse datasets.

We note that due to these limitations the property- and cluster-property tables have not become mainstream in RDF systems.

2.1.2 Graph-based approach

In this section we briefly survey few systems in which the RDF dataset is treated as an edge-labeled graph.

The gStore system [31] focuses on handling wildcard SPARQL queries on disk-resident graphs. In doing so, it assigns to each vertex of the graph (i.e., every RDF resource) a bitstring vertex signature. Each signature is formed based on the nodes' neighborhood in the following way. Each neighbor resource connected with some predicate-labeled edge is encoded with $M + N$ bits, with M bits for the predicate encoding and N bits for the object encoding. The predicate and IRI in object positions are encoded with some hash functions. If the object is literal, its N bits are obtained by first getting the set of 3-grams of that literal and then applying a hash function to that set. The resulting signature of the vertex is disjunction of all its neighbors signatures. Similarly, at runtime a query is mapped to the signature query, and its executing boils down to subgraph matching over

signature graph. Since the latter problem is NP-hard, Zou et al [31] introduced an index-based filtering strategy to reduce its complexity.

An index structure built over signature nodes in gStore is coined the VS-tree. Every leaf node in the VS-tree corresponds to a signature of some vertex in the RDF graph (and all vertexes are represented), and higher level nodes are obtained by applying OR to the children. Such a data structure resembles a S-tree which aims at inclusion queries. An inclusion query roughly corresponds to a single triple pattern matching. A simple S-tree, however, fails to efficiently support joins over several triple patterns since it involves running multiple inclusion queries. The VS-tree therefore additionally introduces super edges between leaves, if the corresponding nodes in the RDF graph are connected. The super edges are propagated to the higher level nodes of the VS-tree and labeled with the bit-wise OR of all the edges between corresponding leaves. Each level of the VS-tree therefore represents the graph in a smaller summarized way. In order to find matchings for the query signature, the VS-tree is traversed in a top-down manner, eliminating at each step impossible matchings by bitwise operations over inner nodes and query nodes, until the leaves are reached.

Udrea et al [28] propose the binary-tree shaped index structured coined GRIN, where every tree node represents a set of nodes from the RDF graph (and hence, an induced subgraph around these nodes). Each inner tree node is associated with the carefully selected "center node" n and a distance d, such that an inner node is a shortcut for all the nodes from the RDF graph which are closer than the distance d from n. The root node represents the entire graph, and the subgraphs corresponding to the inner nodes of the same level do not overlap. Moreover, the subgraph induced by the parent node is the union of two child subgraphs. It is shown that the GRIN index can be constructed fairly efficiently using existing clustering algorithms.

Tree labeling schemes have been extensively studied in context of XML databases aiming at topological relationship between nodes based on their labels. As such, this approach can be extended to store an acyclic RDF graph if the graph is transformed into a tree. Namely, nodes with multiple incoming edges are duplicated, and a new root node above the old one is added [18, 19]. Some of the predicates in RDF graph (that form the Class hierarchy) indeed never form cyclic transitive dependencies. The interval-based indexing of acyclic and cyclic graphs is further analyzed in [7].

2.2 Query Processing and Query Optimization of SPARQL queries

For a given SPARQL query, the RDF engine constructs a query plan for it and executes it. In doing so, the system usually performs the following steps: (i) parse the query and construct the AST, (ii) translate the AST into the query graph, (iii) construct the optimal query plan from the query graph w.r.t. some cost function, (iv) execute the query plan. In this section we describe how these steps are done in modern RDF query engines. Specifically, in Section 2.2.1 we sketch the steps required for turning the query (in the AST form) into the query graph, and discuss different types of query graphs. Section 2.2.2 and Section 2.2.3 then describe query optimization as finding the optimal join ordering based on cost function that uses cardinality estimations. Finally, in Section 2.2.4 we review the two important physical operators, namely the join operator and the path operator. We concentrate here on triple stores since the majority of widely accepted and efficient systems store the data in giant triple tables.

2.2.1 Translating SPARQL queries

Given the SPARQL query, the query engine constructs a representation of it called the *join query graph*. Specifically, a (conjunctive) query is expanded into the set of triple patterns. Every triple pattern consists of variables and literals that are mapped to their IDs in case dictionary encoding is used for triple representation. Then, the triple patterns are turned into the nodes of the join query graph. Two nodes of the graph are connected with an edge iff the corresponding triple patterns share (at least) one variable. Conceptually, the nodes in the query graph entail scans of the database with the corresponding variable bindings and selections defined by the literals, and the edges in the query graph correspond to the join possibilities within this query. The join query graph for a SPARQL query corresponds to the traditional query graph from the relational query optimization, where nodes are relations and join predicates form edges.

Given the query graph, the optimizer constructs its algebraic representation in scans (triple patterns) and join operators, which we call the join tree. This tree along with decisions on the physical level (indexes, types of joins etc) later serves as the execution plan of the query. The join tree can be depicted (hence the name) as a binary tree whose leaves are the triple patterns and inner nodes are joins. The two important classes of join trees are: (a) left-deep (linear) trees, in which every join has one of the triple patterns as input, (b) bushy trees, where no such restriction applies. Naturally, the class of bushy trees contains all left-deep trees.

Using the query graph, one could construct an (unoptimized) plan execution as follows:

1. Create an index scan for every triple pattern, where literals and their positions determine the range of the scan and the appropriate index.
2. Add a join for each edge in the query graph. In case multiple edges connect two nodes, they are turned into selections.
3. In case the query graph is disconnected, add cross products to come up with a single join tree.
4. Add a selection containing all FILTER predicates.
5. Add an aggregation operator if the query has the *distinct* modifier.
6. If dictionary encoding is used, add the ID-to-string mapping on top of the plan.

Additionally, a SPARQL query may contain the disjunctive parts expressed via the UNION and the OPTIONAL clause. The UNION expression returns the union of the bindings produced by its two input pattern groups. The OPTIONAL clause returns the bindings of a pattern group if there are any results, and NULL otherwise. These disjunctive parts of the query are treated as nested subqueries, that is, their nested triple patterns are translated and optimized separately. Then the optimizer combines these subquery plans with the base plan. Namely, for UNION we add the union of the subquery result with the main query, and for OPTIONAL we add the result using the outer join operator.

This sequence of steps results in a *canonical* query plan. Then, the optimizer can employ simple techniques to make it more efficient, like splitting the FILTER condition into conjunction of conditions and pushing these conditions as deep as possible towards the leaves in the join tree. However, the biggest problem for the engine is to order joins such that the amount of intermediate results transferred from operator to operator is minimized.

2.2.2 Join Ordering

For a given query, there usually exists multiple join trees, and the optimizer selects the best one w.r.t. a certain cost function. A typical cost function evaluates the join tree in terms of cardinality of triple patterns and cardinality of intermediate results (i.e., the results of executing subtrees of the join tree) generated during query execution. The cardinality of a triple pattern is the number of triples that match this pattern. In order to estimate the intermediate result size, the notion of join selectivity is introduced. Let t_1 and t_2 be two triple patterns such that the join can be formed between them (i.e. they share at least one variable). We denote their cardinalities with c_1, c_2 respectively. The selectivity of the join between t_1 and t_2 is then defined as the size of the join (i.e., the number of triples satisfying both t_1 and t_2) divided by $c_1 \cdot c_2$. Typically selectivities are estimated by the optimizer using some precomputed statistics. These estimated selectivities along with the (precomputed or estimated) cardinalities of the triple patterns provide the way to estimate the cardinality of intermediate results for the specific join tree. In return, these intermediate size estimations are used to select the join tree with the minimal cost.

The core problem of SPARQL query optimization lies in the cost-based join ordering. The intrinsic properties of RDF and SPARQL require the query optimizer to take the following issues into account while ordering the joins:

- Star-shaped subqueries are very common in SPARQL, so fast generation of optimal plans for them is necessary
- These star-shaped subqueries are connected with long chains, therefore making bushy trees ubiquitous as join plans
- Aggressive indexing employed by the triple store calls for the interesting order-aware plan generation

The state-of-the art solution that addresses all these issues is based on the bottom-up dynamic programming (DP) algorithm [20, 24]. It keeps a DP table of subplans (that correspond to subgraphs of the query graph), each subplan is mapped to its cost and the interesting order. If there are several plans with different order that correspond to the same subgraph of the query graph, they are all kept in the table. The DP algorithms starts populating the table with the triple scans, and at the end of its execution the DP table contains the optimal query plan. In the nutshell, the algorithm works as follows:

1. Seed the plan table with index scans. There are two factors that affect index selection. First, the literals in the triple are handled by the range scan over the corresponding index. Second, another index may produce suitable order for the merge join later on, so all possible index scans are kept at this stage. The optimizer employs the cost-based plan pruning strategy to discard some of these plans.
2. The optimizer evaluates the plans based on the cost function, and prunes equivalent plans if they are dominated by cheaper plans with the same interesting order.
3. Starting from seeds, larger plans are constructed from optimal smaller ones. Since cross products are not allowed, only the plans that correspond to the connected subgraphs of the query graph are considered.

During the process, the algorithm reasons in terms of equivalence classes of variables, and not specific variable values. Variables in different triple patterns are equivalent, if they have the same binding value in the output of the query. This is defined not only by the same name of the variables, but also by FILTER conditions. These equivalence classes later help detect the transitive join conditions.

Figure 2.1: DP table construction

(a) Triple patterns

(?s, hasLongitude, ?long)
(?s, hasLatitude, ?lat)
(?s, type, ?t)

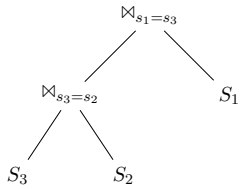
(b) Index scans

S_1=scan[PSO,P=hasLongitude,S=s_1,O=long]
S_2=scan[PSO,P=hasLatitude,S=s_2,O=lat]
S_3=scan[PSO,P=type,S=s_3,O=t]

(c) Joins

$S_1 \bowtie_{s_1=s_2} S_2$, cost=$1.4 \cdot 10^8$
$S_1 \bowtie_{s_1=s_3} S_3$, cost=$1.9 \cdot 10^6$
$S_2 \bowtie_{s_2=s_3} S_3$, cost=$1.9 \cdot 10^6$

(d) Output plan



An example of the DP table construction is given in Figure 2.1 for the query with three triple patterns (Figure 2.1a), that are joined on common subject. Since all these triples have constant predicate, the scan on the PSO index is the cheapest one, given that we later will join them on the subject (Figure 2.1b). Note that in these index scans all s_i are getting different bindings independent of each other. Three join operators (i.e. subplans of the size 2) are derived from the query graph and their cost is determined based on the size of their output (Figure 2.1c). The subplan of size 3 is already the output plan, and it is shown in Figure 2.1d. Since we have used only two joins out of three, the canonical translation would add a selection on a missing join predicate, but in this case we determine that all these s_i are in fact in the same equivalence class, so the existing join conditions are enough.

2.2.3 Cardinality Estimation

So far we have assumed that the optimizer has estimations of sizes of all produced plans readily available. However, obtaining these estimations is a standard (yet extremely complicated) problem in query optimization in database systems. In RDF systems, the correct estimations are hindered by diverse schema-less datasets with multiple implicit dependencies and correlations.

Estimating the cardinalities of individual triples is relatively easy, provided that pre-aggregated binary and unary projections SP,PS,PO,OP,SO,OS and S, P, O are stored (these are typically orders of magnitude smaller than the corresponding full indexes). Then, given the triple pattern, the constants in it define the index that has to be looked at, with these constants as search keys.

Estimating the sizes of joins of two and more triple patterns is much more challenging. Early work [24] builds specialized histograms as an aggregated data structure on top of aggregated indexes of RDF-3X. Originally, all triples with the same two-length prefix are put in separate bucket, then the smallest adjacent buckets are merged until the total structure is small enough. Each bucket is accompanied with a few precomputed values. First, the number of distinct triples, 1- and 2-prefixes are computed, thus allowing to estimate the cardinality of scans. Second, the sizes of joins of tuples from the bucket with the entire table according to all possible (9 in total) join conditions are computed. These sizes provide perfect predictions of the join size only if one triple pattern matches an entire bucket, and the other one matches the whole database.

The method suggested in [27] gathers separate frequencies for every literal and IRI value, and then computes the cardinality of a triple pattern assuming independence of subject, predicate and object. The technique of [17] goes beyond that by selecting graph patterns and gathering statistics about their occurrence in the dataset. There graph patterns are, however, subjects to complex optimization problem. To overcome the computational complexity of [17] and yet go beyond simple statistics, Neumann and Weikum [24] suggested using frequent paths for selectivity estimations. They differentiate between chain and star paths that correspond to shapes of frequent subgraphs in RDF graphs. The paths are ranked by the number of distinct nodes in them, thus avoiding counting nodes twice in cycles. During the query optimization step, the parts of the query that correspond to

frequent paths are estimated using these precomputed statistics, and the rest of the query gets estimates from histogram, assuming independence of different parts of the query.

In the follow-up work [23] the join estimations are improved by the following consideration: in order to estimate the size of the join between two triples, it is enough to know the size of the join between the first triple and the entire dataset, and the selectivity of the second triple. Furthermore, assuming that the first triple has a form (c_1, c_2, v) and the entire table is matched by (s_2, p_2, o_2), the selectivity of the latter join is

$$\frac{|(c_1, c_2, v) \bowtie_{v=s_2} (s_2, p_2, o_2)|}{|(c_1, c_2, v)||(s_2, p_2, o_2)|} = \frac{\sum_{x \in \Pi_v(c_1,c_2,v)} |(x, p_2, o_2)|}{|(c_1, c_2, v)||(s_2, p_2, o_2)|},$$

where the summation is done by the nested-loop join. These selectivities are precomputed offline for every possible choice of the constant (or constants, if the triple pattern has only one variable).

Neumann and Moerkotte [22] suggested the data structure coined *characteristic set* aimed at improving cardinality estimates without independence assumption for star-shaped subqueries. The characteristic set for a subject s is defined as $S_c(s) = \{p|\exists(s, p, o) \in G\}$ for the RDF graph G. Essentially, the characteristic set for s defines the properties (attributes) of an entity s, thus defining its class (type) in a sense that the subjects that have the same characteristic set tend to be similar. The authors of [22] note that in real-world datasets the number of different characteristic sets is surprisingly small (in order of few thousands), so it is possible to explicitly store all of them with their number of occurrences in the RDF graph.

Special care has to be taken of the situation when one objects in the star-shaped query is constant. Then, we can first estimate the cardinality of the star with unbound object and then multiply it by conditional selectivity of the object (i.e., the proportion of triples with that object given the fixed value of predicate). The conditional selectivity avoids independence assumption between an object and a predicate. In case several objects in the star are bound, only the most selective one is taken into account, so we do not underestimate the result size assuming object value independence.

In order to apply characteristic sets to query optimization, the system reasons in terms of the RDF query subgraph. Given the subplan whose cardinality needs to be estimated, the optimizer reconstructs the corresponding subgraph of the RDF query graph, and covers it with available star-shaped synopses based on stored characteristic sets. This way, star-shaped parts of the query will be estimated correctly (which in most cases significantly improves the quality of the plan), and the rest of the query is estimated using traditional independence assumption and available histograms.

2.2.4 Physical operators: joins, path traversals

Even after the optimizer has found the optimal plan for a query with regards to the cost model, the execution of it still faces difficulties, especially for heavy-weight operators like joins and path traversals. Both of these operators may require scanning large portions of the indexes even if the result set is very small.

An important characteristic of SPARQL queries is that individual triple patterns and even subqueries tend to be very unselective, even if the result size of the entire query is small. Therefore, index scans for triple patterns will touch a large portion of the data. To address this problem, sideways information passing (SIP) has been suggested in [23] as a lightweight run-time technique that allows to pass filters on subjects, predicates, objects between scans and joins to speed up their execution. The authors observe that, in order for this join to produce the result, the S values in the first index scan have to find the matching S values in the second index scan. By the pipelined nature of the query executions, the two scans are executed in parallel and can therefore exchange the information about their S value. In particular, if one of the scans encounters large gaps in its S value, the other scan can large parts of its index. Naturally, this approach extends for exchanging information with other scans and merge joins in the operator tree.

In order to enable sideways information passing, the variables in the triple patterns (and the corresponding operator in the join tree) are divided into equivalence classes. Two variables a and b are equivalent, if they appear in some join condition of a form $a = b$. This can happen because of two triple patterns sharing one variable, or due to the FILTER condition. For each equivalence class, the domain information of its variable is kept in

the shared-memory data structure, enabling passing information between different scans and joins that use that variable. It is noted in [23] that SIP speeds up the execution of complex queries by an order of magnitude.

It has been observed [1, 29] that the path traversal on RDF graphs presents challenges for the triple stores and vertically partitioned storages. This stems from the fact that the path traversal boils down to (multiple) Subject to Object self-joins on a large table. One approach to tackle this is to pre-select certain path expressions and materialize them [1, 14]. However, this solution lacks generality since materializing all possible paths is clearly not possible. The aggressive indexing of RDF-3X and Hexastore makes the problem easier, since the multiple joins along the path can be executed as efficient merge-joins supported with sideways information passing. Additionally, in [8] it is shown that the breadth-first search traversal can be speed up significantly by leveraging joins in RDF-3X and improving data locality via dictionary re-encoding.

2.3 Choke point analysis

In this section we will analyze and classify the choke points. By choke points we understand the technical difficulties inherent to RDF data and SPARQL queries, with which real systems need to cope in order to efficiently support complex queries on real-world data. Although some of these difficulties are (to a certain degree) already resolved by some systems and research prototypes, we believe such technical problems should be considered when designing good benchmarks.

We divide the choke points in two large classes. First one, the Query Optimization choke points, considers the challenges that the query optimizer faces when constructing the logical plan for the given SPARQL query. It therefore mostly considers the algorithmic challenges presented by complex SPARQL queries. The second part, Cardinality Estimation choke points, describes the broad set of issues brought by the real-world data. Such data usually comes with implicit dependencies (correlations), which break the simplistic assumptions that optimizer makes in order to estimate the selectivity of parts of the query. This leads to suboptimal decisions on both the logical level (i.e., finding the right join ordering) and the physical level (i.e., choosing appropriate physical operators).

The rest of this section is organized as follows. We first briefly define the main concepts of query optimization which will be used throughout rest of the section. Then we describe the dataset and the RDF system which we used to illustrate the choke points. After that, in Sections 2.3.3 and 2.3.4 we describe the technical challenges of both classes with examples that illustrate them.

2.3.1 Overview of terms and facts from query optimization

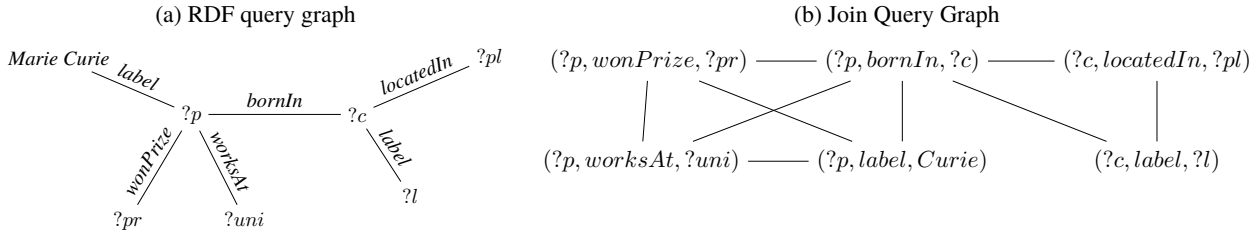
In this section we briefly revisit traditional terminology of query optimization. For a given SPARQL query we define two different types of *query graphs*: First, the join graph, and second, the RDF subgraph. The RDF query graph is the graph where nodes represent variables from the SPARQL query, and the triple patterns form edges between the nodes. Intuitively, the RDF query graph describes the pattern that has to be matched against the dataset. In the join query graph the nodes are the triple patterns of the corresponding SPARQL query, and two nodes are connected with an edge iff they share a variable. Edges in the join query graph therefore correspond to the join possibilities within this query. The join query graph thus corresponds to the traditional query graph in relational query optimization, where nodes are relations and join predicates form edges. It has been shown in literature that RDF graphs of SPARQL queries are convenient for cardinality reasoning, while join ordering algorithms operate on top of join graphs. An example of SPARQL query and corresponding query graphs is given in Query 2.1 and Figure 2.2

We can already see from the example above that, in terms of RDF query graph, a typical SPARQL query consists of several star-shaped subqueries connected via chains. Each star-shaped subquery corresponds to some entity whose properties are specified by the predicates on the edges that form the star. In our example, these subqueries are formed around variables $?p$ and $?c$, they describe a person and a city, respectively. Translated to the join graph view, a typical SPARQL query is a sequence of clique-shaped subqueries connected with chains.

Given the query, the optimizer constructs its algebraic representation in triple patterns and join operators, which we call the join tree. It can be depicted (hence the name) as a binary tree whose leaves are the triple patterns and inner nodes are joins. The two important classes of join trees are: (a) left-deep (linear) trees, in which every join has one of the triple patterns as input, (b) bushy trees, where no such restriction applies. Naturally, the class of bushy trees contains all left-deep trees.

For a given query, there usually exists multiple join trees, and the optimizer selects the best one w.r.t. a certain cost function. A typical cost function evaluates the join tree in terms of cardinality of triple patterns and cardinality of intermediate results (i.e., the results of executing subtrees of the join tree) generated during query execution. The cardinality of a triple pattern is the number of triples that match this pattern. In order to estimate the intermediate result size, the notion of join selectivity is introduced. Let t_1 and t_2 be two triple patterns such that the join can be formed between them (i.e. they share at least one variable). We denote their cardinalities with c_1, c_2 respectively. The selectivity of the join between t_1 and t_2 is then defined as the size of the join (i.e., the number of triples satisfying both t_1 and t_2) divided by $c_1 \cdot c_2$. Typically selectivities are estimated by

Figure 2.2: Query graphs of a SPARQL query



the optimizer using some precomputed statistics. These estimated selectivities along with the (precomputed or estimated) cardinalities of the triple patterns provide the way to estimate the cardinality of intermediate results for the specific join tree. In return, these intermediate size estimations are used to select the join tree with the minimal amount of intermediate results.

As the join tree specifies the ordering of join operations among triple patterns, we refer to the problem of finding the best join tree as the join ordering problem.

Query 2.1: Simple SPARQL query

```
SELECT *
WHERE {
        ?p rdfs:label "Marie Curie"@eng.
        ?p yago:hasWonPrize ?pr.
        ?p yago:worksAt ?uni.
        ?p yago:wasBornIn ?c.
        ?c yago:isLocatedIn ?pl.
        ?c rdfs:label ?l
}
```

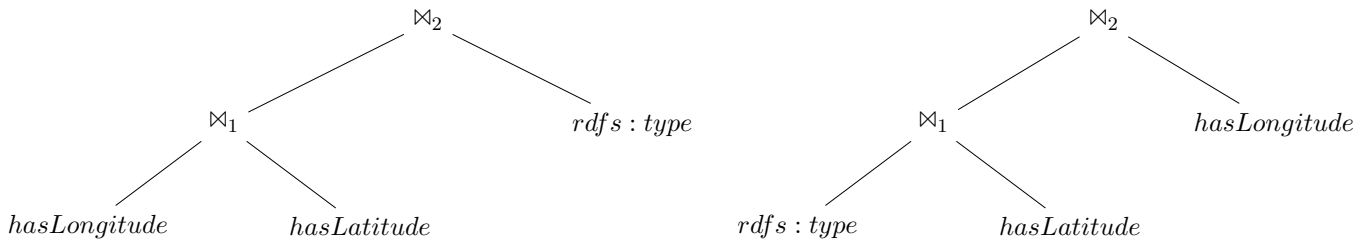
2.3.2 Experiment setup and datasets

Every choke point will be accompanied by an example of SPARQL query and in most cases the experimental facts which we obtained with the our own RDF-3X system [24]. We use YAGO [11], a knowledge base harvested from Wikipedia, as the dataset. The current version of it, YAGO2S, contains around 100 million statements about 10 million facts and therefore presents an example of the real world large dataset.

2.3.3 Query optimization

Join ordering for star queries

As the star-shaped (clique in relational terms) queries and subqueries are very common, the ability of the optimizer to quickly find the optimal plan for them is crucial. A naive and fast algorithm for finding the best join ordering for such queries would order the triple patterns that form the star in order of increasing cardinality. However, such strategy misses the fact that two unselective triple patterns may form a very selective join, or the other way round. Consider Query 2.2 as an example. There, ordering triple patterns by their cardinalities leads to a join tree depicted in Figure 2.3a, while the optimal plan is given in Figure 2.3b. Although both *hasLongitude* and *hasLatitude* predicates are more selective than *rdfs:type*, the size of their join is approximately 20 times bigger than the cardinality of any of the two (this means that on average, one entity in YAGO has 20 different coordinates). On the other hand, not every entity with the *rdfs:type* predicate also has the coordinates, so starting with *rdfs:type* yields a significantly better plan, although *rdfs:type* is quite unselective itself. The resulting runtimes of the two plans are 700 ms for the optimal plan and 2100 ms for the suboptimal one.



(a) Suboptimal Plan: $|\bowtie_1| = 140$ Mln (b) Optimal Plan: $|\bowtie_1| = 1.9$ Mln

Figure 2.3: Query Plans for Query 2.2

Query 2.2: Star Query

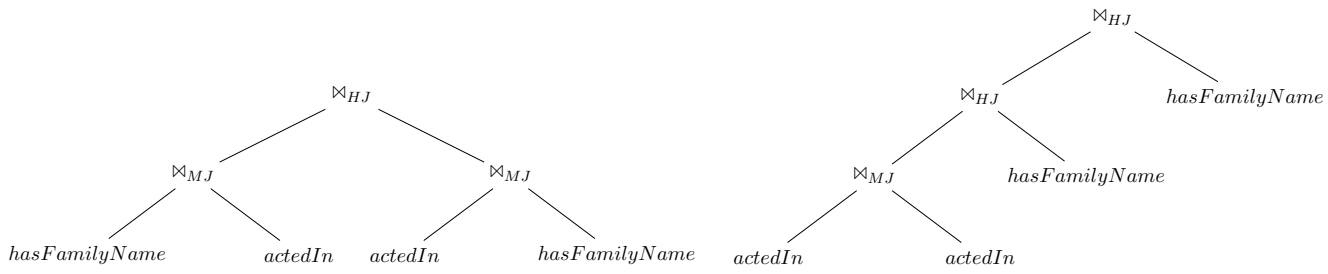
```
SELECT ?s ?t WHERE {
        ?s yago:hasLongitude ?long.
        ?s yago:hasLatitude ?lat.
        ?s rdfs:type ?t.
}
```

Large search space

Dynamic programming algorithm for join ordering, employed by RDF-3X and other systems, successfully overcomes the problem mentioned in the previous choke point. Namely, it always finds the optimal join ordering w.r.t. the given cost function. However, Dynamic Programming comes at a cost: as it finds the exact solution of an NP-hard problem, its complexity is exponential in the query graph size. Thus, for relatively large queries finding an exact join ordering may take prohibitively long time. Consider Query 2.3 that finds information about Thomas Mann and related movies and operas. The query contains 23 triple patterns, so the search space for the Dynamic Programming algorithm is in order of $2^{23} \approx 8 \cdot 10^6$ plans to evaluate. In this case finding the optimal join ordering for Query 2.3 takes 30 seconds while execution of the optimal plan is done within 200 ms, i.e. two orders of magnitude faster!

Query 2.3: Large Search Space

```
SELECT *
WHERE {
        ?s rdfs:type yago:German_people_of_Brazilian_descent.
        ?s rdfs:type yago:Nobel_laureates_in_Literature.
        ?s rdfs:type yago:Technical_University_Munich_alumni.
        ?s yago:diedIn ?place.
        ?place yago:isLocatedIn ?country.
        ?s yago:created ?piece.
        ?piece yago:linksTo ?movie.
        ?movie rdfs:type yago:1970s_drama_films.
        ?director yago:directed ?movie.
        ?director yago:hasWonPrize ?prize.
        ?director rdfs:label "Luchino Visconti di Modrone".
        ?piece yago:linksTo ?city.
        ?city yago:isLocatedIn yago:Italy.
        ?piece yago:linksTo ?opera.
        ?opera rdfs:type yago:wikicategory_Operas.
        ?opera rdfs:type yago:English-language_operas.
        ?composer yago:linksTo ?opera.
```



(a) Optimal plan (b) Linear tree

Figure 2.4: Bushy vs. Linear trees

```
        ?composer rdfs:type yago:English_classical_pianists.
        ?composer rdfs:type yago:English_pacifists.
        ?s yago:influences ?person2.
        ?person2 rdfs:type yago:Animal_rights_advocates.
        ?s yago:created ?piece3.
        ?piece3 yago:linksTo yago:New_York_City.
}
```

Bushy trees and linear trees as query plans

While considering only left-deep trees allows for easy heuristics (see Join ordering for the clique queries) and guarantees that only one intermediate result is generated at any point of query execution, sometimes this strategy misses the optimal plan [13]. A bushy tree plan can be preferred over linear trees due to the following reasons: (a) it reduces the amount of intermediate results, and (b) it allows for asynchronous execution of its subtrees. Consider for example Query 2.4 that finds all the actors that played in the same movie together. The optimal tree w.r.t. the cost function happens to be bushy and is depicted in Figure 2.4a, while the best plan found among linear trees only is presented in Figure 2.4b. As usual, predicates there depict corresponding triple patterns, and joins are performed on common variables. In particular this means that in the optimal bushy tree the subtrees are formed by the triple patterns with a common variable. Note that, in addition to the two advantages of bushy trees mentioned above, this particular bushy tree also makes better use of existing indexes by executing joins in both subtrees as Merge Joins, as opposed to the linear tree where all the joins except the first one are hash joins. Here, the bushy plan results in execution time of 160 ms whereas the linear plan yields 730 ms running time, i.e. it is almost 5 times slower.

Query 2.4: Bushy vs. Linear trees

```
SELECT DISTINCT ?A ?b ?name1 ?name2
WHERE {
     ?A yago:hasFamilyName ?name1.
     ?A yago:actedIn ?film.
     ?b yago:hasFamilyName ?name2.
     ?b yago:actedIn ?film.
}
LIMIT 100
```

Queries with OPTIONAL clause

The OPTIONAL clause in SPARQL corresponds to a left outer join in SQL. Similarly to the relational algebra, this operator can not be freely reordered with other joins, since the respective associative laws do not hold.

Therefore, the optimizer may put the OPTIONAL triples on the top of the join tree, that is, defer their execution until the very end. This behavior will not be optimal for the situation when there is a filtering expression on the variables from OPTIONAL. For instance, in Query 2.5 the optional clause should be pushed down as much as possible since there is a filtering condition on its variable. Pushing the clause down in the join tree would thus significantly reduce the amount of intermediate results. For recognizing the situations where the OPTIONAL clause can be pushed down, the optimizer should employ a variant of Dynamic Programming on hypergraphs as in [21].

Query 2.5: OPTIONAL and FILTER conditions

```
SELECT *
WHERE {
        ?s yago:hasFamilyName ?familyName.
        OPTIONAL {?s yago:hasGivenName ?name.}
        ?s yago:wasBornOnDate ?date.
        FILTER REGEX(?name, "^ali", "i") ||
               REGEX(?familyName,"^sm","i").
}
```

2.3.4 Cardinality Estimation

Until now we have assumed that optimizer is able to calculate exactly the amount of intermediate results produced by a query. However, this is frequently not the case: clearly, finding the exact size for an arbitrary query is only possible after executing it. So, the optimizer has to resort to heuristics that quickly estimate the size of the given join (sub)tree. Among the key assumptions that help doing that is the independence and uniformity assumption. That is, for simplicity the optimizer assumes that values of attributes are uniformly distributed, and values of two different attributes are independent, so that $Prob(A = a_1 \&\& B = b_1) = Prob(A = a_1) \cdot Prob(B = b_1)$.

In the rest of this section we will see how this assumption is violated by real-world datasets, and demonstrate example of queries for which the correct selectivity estimations are crucial and at the same time the typical assumptions of the optimizer do not yield the optimal plan.

Correlated data

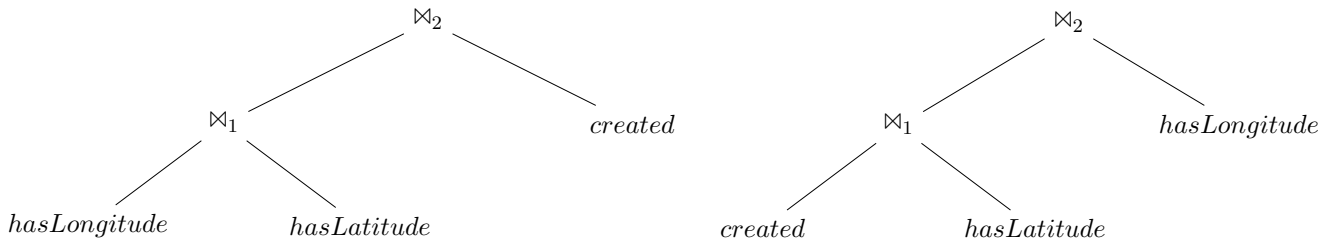
The query optimizer assumes that values of attributes are uniformly distributed, and values of two different attributes are independent, so that $Prob(A = a_1 \&\& B = b_1) = Prob(A = a_1) \cdot Prob(B = b_1)$. In reality, however, the values of attributes frequently depend on each other (the so-called *value correlations*): for example, the name and the country of origin of a person are strongly correlated. It has been shown that uniformity and independence assumptions lead to exponential growth of selectivity estimation errors when the number of joins in the query grows [12]. This effect only becomes more severe in RDF systems since the number of joins in the SPARQL query is larger than in the corresponding SQL query.

Another, RDF-specific, type of correlation is *structural correlation*. As an example, consider triple patterns (`?person`, `<hasName>`, `?name`) and (`?person`, `<hasAge>`, `?age`). Clearly, the two predicates `<hasName>` and `<hasAge>` almost always occur together (i.e., in triples with same subjects), so the selectivity of the join of these two patterns on variable `?person` is just the selectivity of any of the two patterns, say, 1e-4. The independence assumption would force us to estimate the selectivity of the join as $sel(\sigma_{P=hasName}) \cdot sel(\sigma_{P=hasAge})$=1e-8, i.e. to underestimate the size of the result by 4 orders of magnitude!

The combination of the two types of correlations is also quite frequent: Consider an example of triple pattern (`?person`, `<isCitizenOf>`, `<United_States>`) over the YAGO dataset. Now, the individual selectivities are as follows:

$$sel(\sigma_{P=isCitizenOf}) = 1.06 * 10^{-4}$$

$$sel(\sigma_{O=United\_States}) = 6.4 * 10^{-4}$$



(a) Suboptimal Plan: $|\bowtie_1| = 140$ Mln (b) Optimal Plan: $|\bowtie_1| = 14$ K

Figure 2.5: Query Plans for Query 2.6

while combined selectivity is

$$sel(\sigma_{P=isCitizenOf \wedge O=United\_States}) = 4.8 * 10^{-5}$$

We see that both $P =$<isCitizenOf> and $O =$<United_States> are quite selective, but their conjunction is in three orders of magnitude less selective than the mere multiplication of two selectivities according to the independence assumption. The value of the predicate (which corresponds to the "structure" of the graph) and the value of the object here are highly correlated for two reasons. First, the data in the English Wikipedia is somewhat US-centric, and therefore almost half of the triples with P=<isCitizenOf> are describing US citizens. Second, the <isCitizenOf> nearly requires the object to be a country, demonstrating a structural correlation between fields P and O.

Cardinality misestimations: star queries

We have already seen that for star queries, simple ordering of triple patterns by their cardinalities does not yield the optimal plan. However, even the Dynamic Programming approach does not find the cheapest plan if cardinality estimation is guided by the independence assumption, since the optimizer will tend to underestimate intermediate results of the query.

Let us consider Query 2.6. Here, the optimizer under independence assumption misses the fact that entities with *yago:hasLatitude* predicates always have *yago:hasLongitude*, that is the selectivity of the corresponding join is 1. On the other hand, very few entities with *yago:created* attribute have their geographical entities specified (because *yago:created* characterizes people, companies, and places, and geographical coordinates make sense only for places). Therefore, joining first the triple pattern with *yago:created* with either one of the rest leads to a significantly better query plan with the execution time of 20 ms (as opposed to 65 ms for the former plan created with the predicate independence assumption). The two plans are depicted in Figure 2.5.

Query 2.6: Cardinality misestimation: star query

```
SELECT *
WHERE {
        ?s yago:created ?product.
        ?s yago:hasLatitude ?lat.
        ?s yago:hasLongitude ?long
}
```

Cardinality misestimations: complex queries

Naturally, misestimations from the star-shaped subqueries can influence the decisions made on the ordering of different subqueries. In some cases, the unselective but very underestimated subquery is pushed down to the bottom of the join tree, thus producing a large amount of intermediate tuples and influencing the performance of all subqueries higher in the join tree. As an example, consider Query 2.7, for which the suboptimal plan

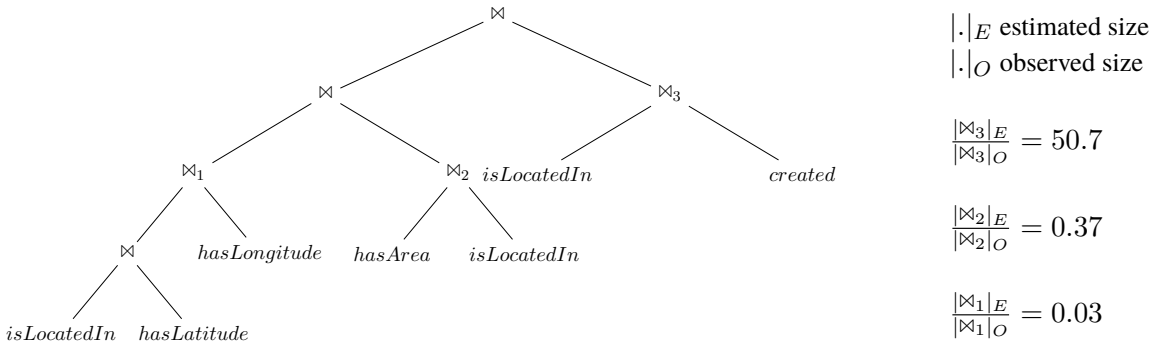


Figure 2.6: (Suboptimal) Plan for Query 2.7

along with the cardinality misestimation ratios are given in Figure 2.6. The problem there is the subtree marked $\bowtie_1$ which is pushed down in the join tree. The optimizer makes a mistake of assuming all predicates to be independent, thus misestimating the size of $\bowtie_1$ by an order of magnitude. On the other hand, the size of the subtree marked $\bowtie_3$ is heavily overestimated. Swapping these two subtrees in the join tree results in a plan with the runtime of 220 ms as opposed to the suboptimal 780 ms runtime.

Query 2.7: Cardinality misestimation: complex query

```
SELECT * WHERE {
        ?s yago:isLocatedIn ?place.
        ?s yago:created ?brand.
        ?place yago:hasArea ?area.
        ?place yago:isLocatedIn ?country.
        ?country yago:isLocatedIn ?location.
        ?country yago:hasLongitude ?long.
        ?country yago:hasLatitude ?lat
}
```

Cardinality estimations and hash joins

Until now we have considered the cases when correct cardinality estimations help building an optimal logical plan, i.e. a join tree. However, reasoning based on cardinality is also important for getting the physical plan right. Namely, constructing the hash join operator between two subtrees usually involves putting the more selective subtree into the build part of the join, and the unselective one into the probe part. An example of such decisions is presented in Query 2.8 that asks for all people born in one country and graduated from a university in another country. There, depending on specific countries, one part of the plan is used in the build phase and another in the probe phase. Consider using a highly unselective (in the context of this query) parameter like USA and a very selective one like Malta. The situation is complicated by the fact that the triples containing parameters *(?place,isLocatedIn,_)* and *(?uni,isLocatedIn,_)* are not joined themselves, but rather connected to the two triples to be joined, so an accurate estimation of the cardinalities of corresponding chains is required to make the right choice.

Query 2.8: Cardinality estimations for hash joins

```
PREFIX yago: <http://yago-knowledge.org/resource/>
SELECT *
WHERE {
        ?s yago:wasBornIn ?place.
        ?place yago:isLocatedIn %country_1%.
        ?s yago:graduatedFrom ?uni.
```

```
        ?uni yago:isLocatedIn %country_2%.
}
```

Cardinality estimations and path queries

Similar considerations about cardinalities should be used in constructing a physical operator for the property path finding. There, a traversal over edges satisfying the expression in the property path expression should be started with the more selective part of the triple pattern (subject or object). We consider a scheme of Query 2.9 as an example of decisions to be made.

If the first triple pattern has the form $(?city, label, ?label)$ and the last one has the form $(?c, type, yago : Baltic\_States)$, then the execution should start with the more selective part, i.e. with the latter triple pattern. If, however, the first triple pattern is $(?city, type, yago : Host\_cities\_of\_the\_Summer\_Olympic\_Games)$ and the last one is $(?c, type, yago : populated\_place)$, then the first triple is more selective and has to be taken as the starting point of traversal.

Query 2.9: Cardinality estimations for path queries

```
PREFIX yago: <http://yago-knowledge.org/resource/>
SELECT *
WHERE {
        ?city %property_1% %attribute_1%.
        ?city yago:isLocatedIn* ?c.
        ?c rdfs:type %attribute_2%.
}
```

2.3.5 Choke points in Graph Databases and Graph Query Languages

In this section we briefly describe how our classification of choke points applies to the case of graph databases and declarative graph query languages. There is no single standard query language adopted by all vendors in the area; here we use Neo4j's Cypher query language.

Join ordering

The `MATCH` operator in Cypher expresses path matching in a graph: for a given expression describing the path labels and/or start and end nodes, it returns all the paths satisfying the expression. Clearly, it corresponds to joins in SQL or SPARQL, where the join predicate is: "two nodes are connected via the given path". If there are several path expressions in the MATCH clause, the query engine is faced with the problem of ordering them – a problem equivalent to the join ordering problem in the RDF and relational databases.

Consider for example the Query 2.6 expressed in Cypher (Query 2.10):

Query 2.10: Path ordering in Cypher

```
START n=node(*)
MATCH (lat)-[:hasLatitude]-(n)-[:created]-(product),
      (n)-[:hasLongitude]-(long)
RETURN n, product, lat, long
```

Here, the query engine has to order three path expressions (with `hasLatitude`, `created` and `hasLongitude` edge labels, respectively), where the `:hasLatitude` and `:created` one-hop paths are subexpressions of the first path in the `MATCH` clause.

Cardinality estimation

Similarly to the SPARQL case, the graph query optimizer also needs to estimate the selectivity of various subqueries in order to find the optimal ordering for paths from the MATCH clause. Consider as an example Query 2.11, which is a Cypher reformulation of Query 2.9:

Query 2.11: Cardinality estimation in Cypher

```
START city=node(*)
MATCH  (city)-[%property_1%]-(%node_1%),
       (city)-[:isLocatedIn*]-(c),
       (city)-[:type]-(%node_2%)
RETURN city, c
```

Here, the cardinalities of the three path expressions, and the result sizes of their combinations, have to be estimated. Two of the path expressions are one-hop paths with a given end node and a given edge predicate, and the third expression describes the variable length path in the graph along the `isLocatedIn` edges.

Unfortunately, the current Neo4j engine does not perform any query optimization of this kind. We believe that constructing the graph DB benchmark that covers the two classes of choke points will encourage community and vendors to consider the cost-based query optimization for graph databases.

3 Conclusion

In the first part of this report we have surveyed the state-of-the art of centralized RDF indexing and querying. The survey is organized by two main issues of the RDF systems design: indexing large RDF datasets and efficiently querying them. The indexing part presents an overview of a wide variety of approached towards storing and indexing RDF graphs, while in the querying part we concentrate on issues related to cost-based query optimization in triple stores, as they are the most mature and popular representatives of RDF systems.

The second part of the report analyzes and illustrates technical challenges that the optimizers need to solve while constructing optimal plans. Namely, we distinguish between the two classes of problems. First, the algorithm-based challenges (NP-hardness of the problem, large search space for complex SPARQL queries, different types of join trees and non-associative operators like OPTIONAL). Second, the data-driven problems of cardinality estimations (estimations for simple star-shaped subqueries and complex queries, influence of correct estimations on physical operators). These problems will serve as a foundation for our SPARQL query benchmark. The two classes of challenges are general and apply to the case of graph databases and graph query languages as well.

REFERENCES

[1] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *The VLDB Journal*, 18(2):385–406, February 2009.

[2] Medha Atre, Vineet Chaoji, Mohammed J Zaki, and James A Hendler. Matrix "Bit"loaded: A Scalable Lightweight Join Query Processor for RDF Data. In *WWW*, 2010.

[3] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient sql-based rdf querying scheme. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 1216–1227. VLDB Endowment, 2005.

[4] Orri Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.

[5] Orri Erling and Ivan Mikhailov. Rdf support in the virtuoso dbms. In *CSSW*, pages 59–68, 2007.

[6] George H.L. Fletcher and Peter W. Beck. Scalable indexing of RDF graphs for efficient join processing. *Proceeding of the 18th ACM conference on Information and knowledge management - CIKM '09*, page 1513, 2009.

[7] Tim Furche, Antonius Weinzierl, and François Bry. Labeling RDF Graphs for Linear Time and Space Querying. In Roberto de Virgilio, Fausto Giunchiglia, and Letizia Tanca, editors, *Semantic Web Information Management*, pages 309–339. Springer Berlin Heidelberg, 2010.

[8] Andrey Gubichev and Thomas Neumann. Path query processing on very large rdf graphs. In *WebDB*, 2011.

[9] Stephen Harris and Nicholas Gibbins. 3store: Efficient bulk rdf storage. In *PSSS*, 2003.

[10] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. Yars2: a federated repository for querying graph structured data from the web. In *Proceedings of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference*, ISWC'07/ASWC'07, pages 211–224, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] Johannes Hoffart, Fabian M, Suchanek Klaus Berberich, Gerhard Weikum, Johannes Hoffart, Klaus Berberich, Gerhard Weikum, Fabian M. Suchanek, and Inria Saclay. Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Commun. ACM*, page 2009.

[12] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD*, 1991.

[13] Yannis E. Ioannidis and Younkyung Cha Kang. Left-deep vs. bushy trees: an analysis of strategy spaces and its implications for query optimization. In *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, SIGMOD '91, pages 168–177, New York, NY, USA, 1991. ACM.

[14] Youn Hee Kim, Byung Gon Kim, Jaeho Lee, and Hae Chull Lim. The path index for query processing on rdf and rdf schema. In *Advanced Communication Technology, 2005, ICACT 2005. The 7th International Conference on*, volume 2, pages 1237 –1240, 0-0 2005.

[15] Justin J. Levandoski and Mohamed F. Mokbel. RDF Data-Centric Storage. *2009 IEEE International Conference on Web Services*, 1(d):911–918, July 2009.

[16] Li Ma, Zhong Su, Yue Pan, Li Zhang, and Tao Liu. Rstar: an rdf storage and query system for enterprise resource management. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, CIKM '04, pages 484–491, New York, NY, USA, 2004. ACM.

[17] Angela Maduko, Kemafor Anyanwu, Amit Sheth, and Paul Schliekelman. Estimating the cardinality of rdf graph patterns. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 1233–1234, New York, NY, USA, 2007. ACM.

[18] Akiyoshi Matono, Toshiyuki Amagasa, Masatoshi Yoshikawa, and S Uemura. An indexing scheme for RDF and RDF schema based on suffix arrays. *Proceedings of SWDB*, 2003.

[19] Akiyoshi Matono, Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. A path-based relational rdf database. In *Proceedings of the 16th Australasian database conference - Volume 39*, ADC '05, pages 95–103, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[20] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *VLDB*, pages 930–941, 2006.

[21] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 539–552, New York, NY, USA, 2008. ACM.

[22] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *ICDE*, pages 984–994, 2011.

[23] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large rdf graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 627–640, New York, NY, USA, 2009. ACM.

[24] Thomas Neumann and Gerhard Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113, February 2010.

[25] Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. Column-store support for rdf data management: not all swans are white. *Proc. VLDB Endow.*, 1(2):1553–1563, August 2008.

[26] Michael Sintek and Malte Kiesel. RDFBroker: A signature-based high-performance RDF store. *The Semantic Web: Research and Applications*, pages 363–377, 2006.

[27] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 595–604, New York, NY, USA, 2008. ACM.

[28] Octavian Udrea, Andrea Pugliese, and V. S. Subrahmanian. Grin: a graph based rdf index. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 2*, AAAI'07, pages 1465–1470. AAAI Press, 2007.

[29] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, August 2008.

[30] Kevin Wilkinson. Jena property table implementation. In *In SSWS*, 2006.

[31] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. gstore: answering sparql queries via subgraph matching. *Proc. VLDB Endow.*, 4(8):482–493, May 2011.