Technische Universität München
Fakultät für Informatik
Lehrstuhl III – Datenbanksysteme

Query Processing and Optimization in Graph Databases

Andrey Gubichev

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Ernst W. Mayr
Prüfer der Dissertation:	Univ.-Prof. Dr. Thomas Neumann,
	Prof. Dr. Peter Boncz,
	VU University Amsterdam, Niederlande
	Prof. Dr. Sihem Amer-Yahia,
	CNRS Grenoble, Frankreich

Die Dissertation wurde am 29.01.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 28.05.2015 angenommen.

Contents

List of Figures

List of Tables

Abstract

Graph data management has received a lot of attention in the last decade, fueled by rapid development of two vertical domains, Linked Data and Social Media. The former is centered around graphs represented as RDF data and queried with SPARQL, while the latter features small-world graphs that fit into the emerging Property Graph model.

This thesis deals with the database aspects of graph processing problems in these two domains. We start with incorporating path and reachability query processing into the state-of-the-art RDF query processing engine, RDF-3X. We then devise a new technique for join ordering and cardinality estimation for general graph pattern matching queries expressed in SPARQL over large RDF datasets. We also present an efficient algorithm for shortest path estimation in small-world graphs and its application to the keyword search on graphs (the Steiner tree problem). In the Property Graph data model, we design the query optimizer's architecture for the popular graph database Neo4j and its query language. Finally, we solve the problem of selecting parameters for query templates as part of our benchmark for the broad class of graph-processing systems.

Zusammenfassung

Graphdatenbanksysteme haben im vergangenen Jahrzehnt viel Aufmerksamkeit erfahren, insbesondere durch die rasante Entwicklung von Linked Data und sozialen Medien. Ersteres basiert auf Graphen, die als RDF-Daten dargestellt und mit SPARQL abgefragt werden. Letzteres beschäftigt sich mit small-world Graphen, die in das aufkommende Property Graph Modell passen.
Diese Arbeit behandelt die Datenbankaspekte von Fragestellungen der Graphanalyse dieser beiden Gebiete. Wir beginnen damit die Bearbeitung von Pfad- und Erreichbarkeitsanfragen in ein aktuelles System, RDF-3X, zu integrieren. Wir entwickeln eine neue Technik zur Join-Ordnung und Kardinalitätsabschätzung für allgemeine Pattern Matching-Anfragen in SPARQL für große RDF-Datenbanken. Zudem präsentieren wir einen effizienten Algorithmus zum Abschätzen kürzester Pfade und ihrer Anwendung für die Schlüsselwort-Suche in Graphen (Steinerbaumproblem).
Im Property Graph-Datenmodell entwerfen wir die Architektur des Anfrageoptimierers für die populäre Graphdatenbank Neo4j und ihre Anfragesprache.
Schließlich lösen wir das Problem der Parameterselektion für Anfragemuster als Teil unseres Benchmarks für Graphdatenbanken.

CHAPTER 1

Introduction

A data management problem becomes a graph problem when it concerns not only analysis of the values, but also discovering and exploiting connections between them. The last decade has seen the rise of interest in graph data management problems, both in academia and in industry. The interest was caused by the rapid spread of graph-shaped data coming from multiple domains with the two main examples being (i) the Linked Data initiative, which exploits cross-domain techniques from data mining, natural language processing and machine learning to construct web-scale knowledge bases, and (ii) Social Media, where the users and their generated content form a quickly growing graph; the availability of large-scale social networks has motivated numerous Social Network Analysis projects. The information needs in the two domains are best expressed in terms of graph problems.

Graph problems can be roughly divided into two large classes. First, there is the class of database problems: how to store and index graphs efficiently, how to support declarative query languages and transactions, as well as the issues of data cleaning, data integration, exploiting modern hardware trends etc. The second class is graph analytics, where the goal is to support a specific graph algorithm over a very large graph. The algorithms include PageRank, shortest paths, centrality computation, graph partitioning; in addition, there is a large number of graph mining and graph modeling problems like predicting links, mining communities and frequent subgraphs, counting triangles and motifs, tracing graph evolution over time etc.

As an indicator of growing importance of graph data management, quite a few graph database systems and graph programming frameworks have been built in academia and industry. The advantage of a specialized database solution lies both in usability and performance. Expressing, understanding and debugging queries for graphs is typically easier in a specialized graph query language (like SPARQL or Cypher). On the other hand, although it is possible to express many graph queries in SQL, the relational query optimizer is likely to fail in producing a good execution plan, since such SQL queries are typically large, contain recursion and expensive self-joins. The specialized query optimizer of

a graph database, on the other hand, may take into account typical graph access patterns and types of queries and develop specialized techniques for their support. Graph databases also usually provide a low-level API access to the data, so that any use case-specific graph algorithm can be implemented on top of the database.
In this thesis we study both the database and graph analytical problems from the Linked Data and Social Media domains. We structure our discussion around the specific problem types, and cover the following topics:

- Shortest path and reachability queries over Linked Data and Social Media datasets. We look at the issues of supporting path queries (expressed in a declarative query language) inside the graph database systems.
- Shortest path and Steiner tree approximation over large graphs. We devise approximate algorithms to accurately estimate shortest paths in large graphs, which can be used as building blocks in various graph analytical problems. As an extension of these algorithms, we design methods for approximate Steiner tree computation, the problem that arises in keyword search over graphs.
- Query optimization for general graph queries, expressed in SPARQL query language. Specifically, we present methods to estimate the cost of the execution plan (*cardinality estimation problem*), as well as a novel technique to explore the plan search space (*join ordering problem*); all of them are tailored to graph databases and have not been considered in the traditional relational database context.
- Query optimization for graph pattern matching queries, expressed in Cypher query language. Here we describe the architecture and design decisions made for the query optimizer of the Neo4j database system.
- Benchmarking graph databases. We address the problem of selecting parameters for benchmark queries to enable fair comparison of the broad class of emerging graph-supporting databases and frameworks.

In the rest of this chapter we provide the background information for the rest of the thesis. We start with describing datasets from Linked Data and Social Networking domains, and give some qualitative characteristics of graphs from these two domains (Section 1.1). We then present two data models that are used to represent and query graphs, namely RDF and Property Graph. We proceed with a description of the RDF-3X query engine, which we use as a backend for our graph query processing discussion throughout the thesis. We finish the Introduction with an overview of the rest of the thesis.

1.1. Graph Use Cases

In this section we present two vertical domains (Linked Data and Social Media) which extensively rely on graph-shaped data, and describe how the data is used there, i.e. what are the queries and database problems that occur in these domains. We also provide some key characteristics of graphs in these two fields that will shape later discussion.

Linked Data

The concept of Linked Data [44] refers to the graph of published entities on the web, where both entities and links are identified with URIs. The entities are often grouped in datasets (such as, for example, various life sciences datasets), and the links connect entities within and across different datasets. Last years have seen an impressive growth of Linked Data on the web, with over 31 billion links (triples in RDF notation) being published online, according to the latest diagram of the Linked Open Data cloud. 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.
Throughout this thesis we will use (among others) the two of these datasets, the knowledge base YAGO2 and the biomedical dataset UniProt.
YAGO2 (Yet Another Great Ontology) [45, 96] is a knowledge repository populated by facts from Wikipedia. It is constructed using infoboxes and category system of Wikipedia, and the Wordnet thesaurus. The knowledge base is essentially an entity-relationship network: the entities of Wikipedia (people, phenomena) are connected with different relationships. In addition to concrete entities, there is a rich type system (e.g., `French_Scientist`), and relationships between the types (e.g., `subclassOf`) and between concrete entities and types (one entity may belong to different types). These relationships are frequently called facts, since the two nodes and an edge between them essentially form a simple statement (e.g., `Louis_Pasteur isA French_Scientist`). As more sophisticated knowledge harvesting techniques were invented over time, the knowledge base has evolved from 5 million facts [96] to over 100 million facts [45].
Another part of the Linked Data cloud is Uniprot [102] — a catalog of information on proteins, partially curated by experts. At the core of it there is a protein sequence collection, where every protein is linked to disease, function, pattern of expression, the proteins it interacts with, and other biological information. Every entity is also annotated with related scientific publications. The crawl of Uniprot used in this thesis contains 845,074,885 distinct facts about proteins.

Once collected, knowledge bases serve as a backend for semantic search engines like NAGA (Not Another Google Answer) [55], which enables precise question formulation to extract entities and relations of interest. As reflected in the name of this exemplary system, the queries that it addresses go beyond the capabilities of modern search engines (i.e., beyond a keyword search on the corpora of web documents). Ideally, the type of questions one should be able to ask a knowledge base varies from *What are the drugs inhibiting this disease?* to *Who was the president of the United States in the year when Barack Obama was born?*.
Such questions, translated to a query language, require a query engine and specific query processing and query optimization techniques to answer them efficiently. In terms of graph operations, these questions range from reachability (*How are Angela Merkel and Richard Wagner related to each other?*) to complex graph pattern matching (*Find a German author whose book inspired an Italian movie set in Venice*).
In this thesis we deal with formal representations of these tasks (using declarative query languages), leaving aside the problem of translating natural language questions into database queries [104]. We address the problems of efficient execution for some graph operations (reachability and shortest paths) on knowledge bases; we also study the problem of query optimization (that is, translation of a declarative query into imperative execution plan) for graph pattern matching queries.

Social Media Graphs

The second class of graph datasets and use cases considered in the thesis is the Social Media graphs, including online social networks and web graphs. Nodes there are typically user accounts or web pages, and links denote either connection between corresponding users (friendship, following etc.), or relationship between web pages (citation, re-posting etc.). One common characteristic that these graphs share is the small distance between any given pair of nodes. The distance between any two nodes is a factor of $\log |V|$, where V is the set of all nodes [103]; it is also a known phenomena that the distance between any two nodes is likely to *decrease* as the social graph grows (i.e., more nodes and connections are added) [61]. Although the original data is owned by the online social network service providers and is not generally available, the graph structure of various networks have been crawled by specialized tools. One of the largest such collections of crawled networks is the Stanford Network Analysis Project[1].
A typical graph processing task over social media graphs is to find connections between two nodes (users), that is to answer the question *What is the shortest*

[1] `http://snap.stanford.edu/`

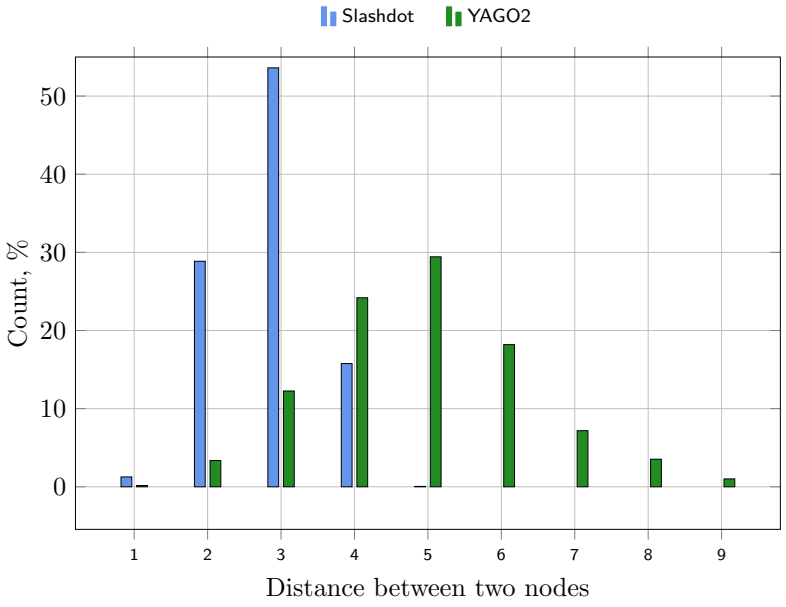


Figure 1.1.: Distance distribution for randomly sampled pairs of nodes (2000 pairs)

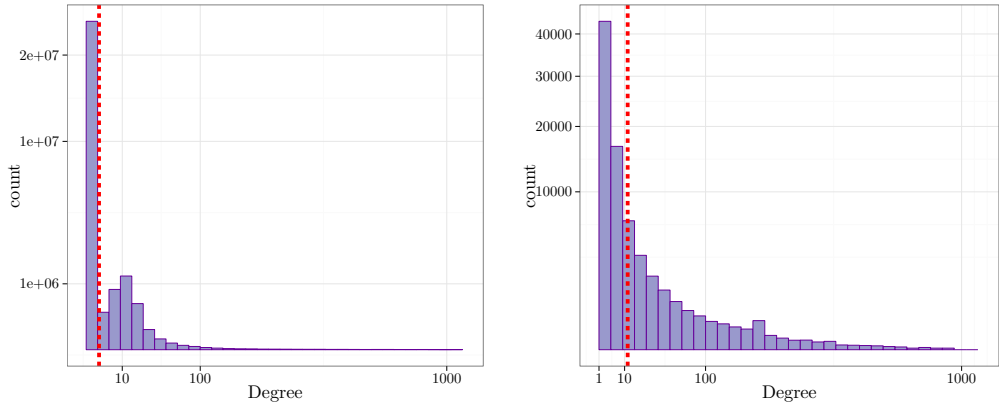
path between two individuals?. As an example, the LinkedIn social network uses the path to a given user as a way to reach out that individual, starting from friends. Shortest paths also serve as a building block for many graph mining tasks, such as diameter and centrality computation. More general problem is to connect several nodes of a graph, such that the induced subgraph minimizes certain objective function. This task occurs, for example, when doing keyword search over graphs. We study both of these problems in the thesis: we give algorithms for approximate shortest path computation, and generalize them for the Steiner tree problem (a variant of the keyword search in graphs).

Graph Characteristics

To illustrate some of the key properties of graph datasets of both classes, we consider two datasets, the knowledge repository YAGO2 and an online social network Slashdot (mined in 2008). We measure and report two network characteristic: the degree distribution of the nodes and the distance distribution for pairs of the nodes.

Distance Distribution. Let $\texttt{dist}(u, v)$ be the length of the shortest path between nodes u and v in graph $G = (V, E)$. Then the *diameter* of G is defined as the maximum length of the shortest path between any node pair:

$$d(G) = \max_{(u,v) \in V \times V} \texttt{dist}(u, v)$$



(a) YAGO2, average degree = 2.1 **(b)** Slashdot, average degree = 11.7

Figure 1.2.: Degree distribution of Slashdot and YAGO2 graphs. The mean degree is marked with the dashed red line

Computing the exact diameter is prohibitively expensive, since it requires computing all-to-all pairs shortest path. In order to get an idea of what the diameter can be, we can draw a sample S of node pairs from $V \times V$ (uniformly, at random) and compute an estimate (lower bound of the diameter)

$$d_e(G) = \max_{(u,v)\in S} \texttt{dist}(u, v)$$

We sample 2000 pairs for YAGO2 and Slashdot, and plot the distribution of $\texttt{dist}(u, v)$ in Figure 1.1. As we see, for both of the datasets the largest distance between nodes from the sample is rather small: all the nodes from the Slashdot sample are within at most 5 hops from each other, and for YAGO2 this number is slightly higher (9 hops). The Slashdot social network (or at least its sample) therefore indeed confirms to the "six degrees of separation" theory of Stanley Milgram.

Degree distribution. For an undirected graph $G = (V, E)$, the degree of a node u, denoted $\texttt{deg}(u)$, is the number of adjacent nodes, i.e. nodes connected to u with just one edge. For a directed graph, there are two properties, out- and in-degree, computed separately for outgoing and incoming edges of u.

We plot the out-degree distribution of YAGO2 and the degree distribution of nodes in Slashdot in Figure 1.2, marking the average degree with the red line and using a logarithmic scale. In both cases it is a power-law distribution: very few nodes have more than 100 neighbors (out-neighbors, respectively).

We are interested in degree distribution because it is a good indicator of the "density" of the graph. Indeed, social networks tend to have high average degrees

(Orkut has an average degree of 38.1, and for LiveJournal it is 14.2). This fact, together with the small diameter of the network, implies that for any two given nodes there likely are a lot of different shortest paths between them. Moreover, if we have one shortest path between two nodes u, v, we can easily construct many other shortest paths by looking at the neighborhoods of the nodes along that path. We exploit this fact and formalize this procedure in Chapter 3 to build indexes for approximate shortest path computation.
On the other hand, YAGO2 is a really sparse directed graph, which is very close to being acyclic. It is a knowledge base, so its entities form an acyclic hierarchy with a few exceptions (mostly caused by existing Wikipedia category system). This "sparsity" allows us to build efficient reachability indexes and support reachability and shortest path queries as part of the SPARQL query language.

1.2. Data models

In order to be stored and queried in a database, the graph dataset needs to be mapped to a specific data model. There are two alternatives for that: RDF and Property Graph data models. This list is of course by no means exhaustive; for instance, some of the industrial solutions map graph datasets onto the relational model and store them in relational database systems. Note that most of the contributions of the thesis do not in fact depend on the specific data model in use. For example, one can use our approximate shortest path algorithms from Chapter 3 even if the graph is mapped to the relational model. Similarly, most of the query optimization problems do not depend on the data model, since they stem from the innate characteristics of the data and not from the way it is represented. To make the presentation concrete, however, we use two specific data models that gained wide acceptance in the community and that are arguably more intuitive for graph-shaped datasets than the relational model.
The formal mapping between two models is presented and illustrated in [41], we adopt our example from there.

1.2.1. RDF

As we have discussed in the previous section, there is a very natural mapping between pairs of connected nodes (together with the edge between them) in the graph of YAGO2 and the facts of the knowledge base: the labels on nodes serve as a *subject* and an *object* of the fact, while the label on the edge expresses a *predicate*. This idea in fact lies behind the Resource Description Format (RDF), which models the graph as a set of triples (`S(ubject)`, `P(redicate)`, `O(bject)`). RDF has been standardized by the World Wide Web Consortium (W3C).

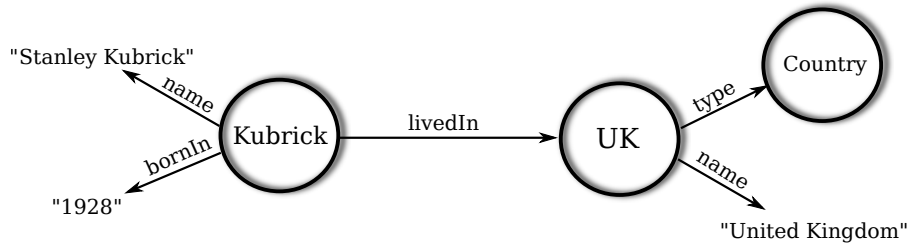


Figure 1.3.: Example of a graph in the RDF data model

According to the W3C standard, the entities are uniquely identified using URIs, and can therefore be used across multiple triples and multiple datasets. URIs can be used in any of the `S`, `P` or `O` positions. Additionally, we can use string literals to identify properties of entities; string literals are therefore only allowed in the `O` position of the triple.
Consider an example graph in Figure 1.3, where we mark URI nodes with circles, and string literals are enclosed in quotes. This graph represents a few facts from the knowledge base, and in RDF it can be expressed as follows:

```
(Kubrick, name, "Stanley Kubrick")
(Kubrick, bornIn, "1928")
(Kubrick, livedIn, UK)
(UK, name, "United Kingdom")
(UK, type, Country)
```

The main power of the RDF format is that it is schema-less: there are no schema requirements that entities need to satisfy in order to be connected to each other. In our example, we can easily add more triples that describe properties of the `UK` entity (including the data from other datasets, e.g. census data), and connect `UK` to other entities. This makes RDF a very attractive option (and the de-facto standard) for storing heterogeneous Linked Data datasets.
Together with the data format, the W3C has standardized the declarative query language SPARQL for it. A very basic SPARQL 1.0 query has a form

```
select ?v1 ?v2 where {pattern1. pattern2. ...}
```

where `pattern1` is a *query pattern*, consisting of `subject`, `predicate` and `object` (each of them is variable or constant). The body of the query is interpreted as a conjunction of patterns, i.e. we are looking for the bindings of variables that satisfy all of the patterns in the query. In database terms this is equivalent to having joins between each pair of patterns that share at least one variable. We will give a concrete example of a SPARQL query and its execution in the database system in Section 1.3.2.
SPARQL is a rich graph pattern matching language; its latest version (SPARQL

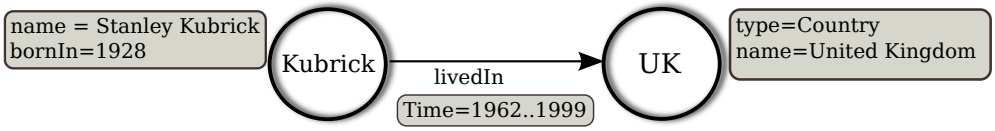


Figure 1.4.: Example of a graph in the Property Graph data model

1.1) includes aggregates, subqueries, abilities to update the data and to express federated queries. In this thesis we will mostly consider the core graph pattern matching features contained in the SPARQL 1.0 (adding reachability and path queries constructs to it in Chapter 2).

1.2.2. Property Graph

The second graph representation format is given by the Property Graph (PG) data model, which gains popularity among graph database users and vendors. Although not standardized, the model is used by popular graph databases such as Neo4j[2] and Sparksee[3].

According to one of the informal descriptions [87], graph in the PG model consists of nodes, edges, and properties. Properties are arbitrary key-value pairs, where keys are strings and values are of arbitrary data types. Both nodes and edges can have multiples properties attached; in addition, an edge between two nodes has a label and a specified direction.

We give an example graph in the PG model in Figure 1.4. It is similar to the RDF example from the previous section, except one thing: the edge between `Kubrick` and `UK` has a property that expresses the time interval in which Stanley Kubrick lived in the UK. Note that the same information is not easy (although not impossible) to express in RDF: one would need to extend triples to quadruples or use reification to represent properties of predicates (statements about statements) [42].

When it comes to querying graphs in the PG model, there is no universally accepted solution, although Neo4j's Cypher seems to be the most popular choice. Note that most of the systems in this area do not have a declarative query language, and provide various imperative APIs for data access instead. In fact, Cypher, to the best of our knowledge, is the only declarative graph query language supported by an available graph database system.

Defining the language formally is out of scope of this thesis; instead, we illustrate its flavor. A simple Cypher query has the following structure:

[2] `http://neo4j.org`
[3] `http://www.sparsity-technologies.com/`

```
MATCH path1, path2,...
WHERE node.property=value
RETURN node
```

where MATCH and WHERE clauses specify the graph pattern as a conjunction of conditions on edges (one- or many-hop paths). The WHERE clause may contain conditions on properties of nodes and edges, and the RETURN clause lists the nodes that the query yields as a result. Here is an example query that returns all the people born in 1928 and living in the UK:

```
MATCH (p:Person)-[:LIVED_IN]->(c:Country)
WHERE p.bornIn=1928 AND c.name='UK'
RETURN p
```

Cypher uses an ASCII-art style of specifying graph patterns, where nodes are enclosed in parentheses, and edges in square brackets. Both nodes and edges may have type conditions (labels), so a pattern for a node (or an edge) that belongs to a fixed `Type` has a form of `(variable:Type)`, where the variable name may be missing for edges. Nodes are connected via dashes into paths; several paths can be specified in a comma-separated list. The result of the query contains nodes of the subgraph that confirms to the union of all path patterns, and satisfies the key-value conditions on the nodes.
We will discuss the difference between Cypher and SPARQL in terms of semantics and from the query optimization perspective in Chapter 5.

1.3. The RDF-3X System

The open-source RDF-3X system is the state-of-the-art research prototype for RDF data management. It is a graph database in that it provides graph pattern matching capabilities by querying the data in SPARQL. This engine will serve as a testbed for further types of graph queries and some advanced query optimization techniques, described in the thesis. Here we provide an overview of the system's architecture and design principles, highlighting the parts that we will use and modify in the thesis. First we describe how the data is stored, then we define the query graph and the SPARQL graph for a SPARQL query – central concepts for the query optimization topics. This discussion is followed by the description of join ordering, cardinality estimation and physical plan construction techniques which are at the core of the RDF-3X's query engine. Parts of this description were previously published in [37] and [34].

1.3.1. Storing RDF data

RDF-3X is a triple store, that means it puts all triples into a giant triple table with the `subject`, `predicate`, `object` (S, P and O) attributes. The engine follows an aggressive indexing strategy and stores all six permutations of S, P and O in clustered B^+-tree indexes: SPO, SOP, PSO, POS, OSP, OPS. 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 $|\{\mathtt{s} \mid (\mathtt{s}, \mathtt{p}', \mathtt{o}') \text{ is in the dataset}\}|$. 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}
```

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 [78] 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.

Note that keeping multiple indexes of the same data is not at all unusual for triple stores, although other solutions are less exhaustive: for example, Virtuoso [27, 28] triple store keeps only two full indexes on quadruples (PSOG, POGS). In addition, three partial indexes SP, OP and GS are kept.

In order to keep space requirements reasonable for excessive indexing, triple stores 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. RDF-3X uses an incremental approach, where every new data item gets an increased internal ID. The dictionaries themselves are stored as B^+-trees.

1.3.2. Query Engine

In order to run declarative queries against stored data, the database has to translate a query into an imperative execution plan, and then execute it. The

engine first translates the query into intermediate algebraic representation (*query graph*), then finds out the ordering of algebraic operations (most importantly, joins). This ordering task is an optimization problem which uses specific cost model for picking the best order. Finally, for the given join order the engine also decides on the specific join and access methods (*physical plan* construction). In this section we discuss all these steps in detail.

Query Representation

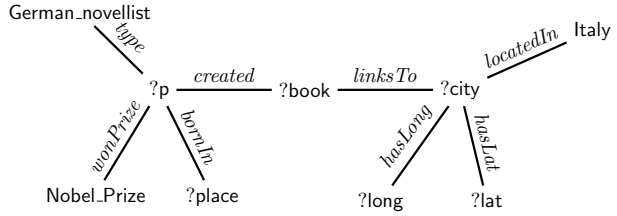
Given a SPARQL query, the query engine starts off with constructing its representation called a *query graph*. Specifically, every triple pattern of the query is turned into a node of the query graph, and two nodes are connected if the corresponding triple patterns share a common variable (or, if there is a FILTER condition relating variables of these two triple pattern). Conceptually, the nodes of the query graph entail scans of the dataset with the corresponding variable bindings, and the edges correspond to the join possibilities within the query. Thus defined, the query graph of a SPARQL query corresponds to the traditional query graph from relational query optimization, where nodes are relations and join predicates form edges.
Another way to represent a SPARQL query is a graph structure that we call a *SPARQL graph*. Its nodes denote variables of the query, while the triple patterns form edges between the nodes. Intuitively, this structure describes the subgraph that has to be extracted from the dataset.
Consider an example SPARQL query and its two representations (the SPARQL graph and the query graph) depicted in Figure 1.5. The query of 8 triple patterns finds a Nobel prize winning author from Germany that had written a book related to an Italian city. The SPARQL graph (Figure 1.5b) describes the pattern that has to be matched against the dataset: we are looking for two star-shaped patterns (around variables `?p` and `?city`) that are connected via the two-hop chain `?p` - `?book` - `?city`. The equivalent query graph (Figure 1.5c) consists of two four-node cliques (induced by variables `?p` and `?city` that appear in four triple patterns each), and a chain between them.

Join Ordering

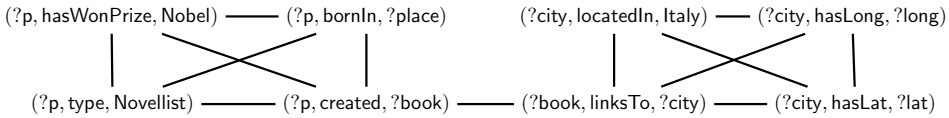
Given the query graph as input, the optimizer returns the query plan, which is defined by the *ordering of joins* between triple patterns, and the type of each join (merge or hash join). A cost-based query optimizer explores the search space of different algebraically equivalent query plans, and selects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

```
select *
where {
    ?p type German_novellist.
    ?p hasWonPrize Nobel_prize.
    ?p bornIn ?place.
    ?p created ?book.
    ?book linksTo ?city.
    ?city isLocatedIn Italy.
    ?city hasLongitude ?long.
    ?city hasLatitude ?lat.  }
```

(a) SPARQL query



(b) SPARQL graph (nodes are variables and constants)



(c) Query Graph (nodes are triples)

Figure 1.5.: Query graphs of a SPARQL query

query plan. For example, the RDF-3X cost functions for merge and hash joins (MJ and HJ, respectively) are defined as follows:

$$cost_{MJ} = \frac{lc + rc}{100}, \qquad cost_{HJ} = 300,000 + \frac{lc}{100} + \frac{rc}{10} \tag{1.1}$$

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

The classical search space exploration technique is Dynamic Programming. Its variant employed by RDF-3X builds the optimal join tree bottom-up, increasing the size of partial solutions at every step. We give a C++-like pseudocode of the algorithm in Listing 1.1. Double `for` loop iterates over (already considered) solutions and tries to combine them into a candidate solution. We keep the mapping between covered triple patterns and candidate solutions (`lookup`). Then, for a candidate solution we check that (i) it was not constructed before, or (ii) it is cheaper than the one in `lookup` (lines 18-19). If either one of the conditions is true, we add the candidate solution to `dpTable`.

Note that this algorithm can be extended by adding another binary operator $\circ$, as long as it is associative (with respect to itself and to the join operator $\bowtie$), that is:

$$(R \circ T) \circ V = R \circ (T \circ V)$$
$$(R \circ T) \bowtie V = R \circ (T \bowtie V)$$
$$(R \bowtie T) \circ V = R \bowtie (T \circ V)$$

in the corresponding domains. The only part to change is `canJoin` function that needs to take into account the $\circ$ operator. We will use this fact in Chapter 2, where we will extend RDF-3X's runtime system with other binary operators.

```
auto dpTable = vector<list<Problem>>();
auto lookup = unordered_map<Set, Problem>();
dpTable[0] = {R1, ..., Rn};
lookup.init(dpTable[0]); // init with leaf plans
for (i = 1; i < dpTable.size(); i++){
  for (j = 0; j < i; j++) {
    for (leftRel in dpTable[j]){
      for (rightRel in dpTable[i-j-1]){
        if (!canJoin(leftRel, rightRel)){
          continue;
        }
        auto rels = SetUnion(leftRel, rightRel);
        // join trees covering leftRel and rightRel
        auto left = lookup[leftRel];
        auto right = lookup[rightRel];
        // candidate solution
        auto P = CreateTree(left, right);
        if (!lookup.count(rels) ||
                    lookup[rels].cost > P.cost){
          // found better solution
          dpTable[i].push_back(P);
          lookup[rels] = P;
        }
      }
    }
  }
}
```

Listing 1.1: Dynamic Programming for Join Order Optimization

Cardinality Estimation

The cost function, mentioned in the previos section, operates on cardinalities (result sizes) of subplans (left and right subplans of a join). Naturally, these numbers are not known at the time of query compilation, and the optimizer needs to estimate them. The state-of-the-art RDF-specific cardinality estimates were introduced by Neumann and Moerkotte [76]. These estimates are computed using *characteristic sets*.

Definition The Characteristic Set for a subject s is defined as $S_c(s) = \{p \mid \exists o : (s, p, o) \in \text{dataset } R\}$. The set of Characteristic Sets for a dataset R is $S_c(R) = \{S_c(s) \mid \exists p, o : (s, p, o) \in R\}$

Essentially, the characteristic set for s defines the properties (attributes) of an entity s, thus defining its class (type) in a sense that the subjects that have the same characteristic set tend to be similar. For example, the class of `Person` in the knowledge base can be defined by the characteristic set {`bornIn`, `livesIn`, `hasName`, `marriedTo`, `hasChild`}. The authors of [76] note that in real-world datasets the number of different characteristic sets is surprisingly small (in order of few thousands), so it is possible to explicitly store all of them with their number of occurrences in the RDF graph.
As an example, consider again the query in Figure 1.5a and suppose that we are to estimate the cardinality of the join between the triple patterns containing `created` and `bornIn` predicates. The natural way to do it is to iterate over all characteristic sets, find those sets that contain both predicates, and sum up their counts. More precisely, the cardinality is estimated as

$$\text{card} = \sum\nolimits_{S \in \{S | S \in S_c(R) \& \{\texttt{created},\texttt{bornIn}\} \subseteq S\}} count(S),$$

where $count(S)$ is the number of occurrences of the characteristic set S, precomputed and stored together with S. Note that this type of computation works for any number of joins within the same star subquery and delivers accurate estimates.
This robust procedure works well for star-shaped SPARQL queries, however it makes Dynamic Programming more expensive, since the cardinality needs to be estimated for *every* considered subplan. In Chapter 4 we extend this data structure for queries of a general shape.

Physical Plan

Since the data is available in several orderings, it is often beneficial to use a merge join as a physical implementation of a join operation. For example, the star subqueries typically are executed using the merge join. If we denote by σ_{order} the table scan in the given *order* (e.g., Object-Predicate-Subject) with the specified constants, the star-shaped subquery around `?p` from Figure 1.5b can be transformed into the following sequence of merge joins:

$$\Big(\Big(\sigma_{OPS}(O = \text{Nobel\_Prize}, P = \text{hasWonPrize}) \bowtie_{MJ} \sigma_{OPS}(O = \text{German\_Novellist}, P = \text{type})\Big) \bowtie_{MJ} \sigma_{PSO}(P = \text{created})\Big) \bowtie_{MJ} \sigma_{PSO}(P = \text{bornIn})$$

The reason for selecting this execution plan are the following. First, in order to determine the *order* of the scan (e.g., OPS in the first scan), the optimizer takes into account the positioning of constants in that scan. Our first scan has

constants in the Object and the Predicate position, therefore the order of the scan is OPS (constants in the Object position are usually more selective, so we start with them). Same holds for the second scan (looking up `O = German_Novelist` and `P = type`). Both of these scans produce results (bindings for the Subject positions) in increasing order, so we can use merge join as a join method. The two remaining scans have only one constant, in the Predicate position. In principle, we can pick any of the two orderings, PSO or POS, but only the PSO order will produce the results ordered by Subject. Since this allows us to keep using merge join (as opposed to building the hash table and doing hash join), this is the preferred order.
Runtime techniques like sideways information passing stress the importance of merge joins even further [77]. Namely, the values of variable `?p` encountered during the index scans can be propagated to other (less selective) index scans participating in the subquery with merge joins, such that the latter could skip most of their pages on disk. Propagation is possible since the merge join keeps the ordering of its input data intact.

1.4. Contributions and Overview

The exposition is structured around the types of the graph processing problems, encountered in Linked Data and Social Media (Small-World) graphs.

- We start with describing efficient mechanisms for path and reachability queries over Linked Data graphs in Chapter 2. There, the graphs are represented in RDF and queries with SPARQL with extensions for paths and reachability expressions. We demonstrate how the RDF-3X system can be extended to efficiently support these kind of queries.

- For cases when exact shortest path discovery is computationally unfeasible (e.g., large dense graphs), we formulate novel approximate algorithms for shortest path estimation in Chapter 3. We also consider a more general problem of connecting multiple nodes of the graph with the smallest possible tree (*Steiner tree problem*), and present a scalable algorithm based on our shortest path estimation methods.

- We then turn our attention to queries more complex than path extraction, such as general pattern matching problems expressed in SPARQL over RDF graphs in Chapter 4. We improve the query optimization capabilities of RDF-3X in two ways: (i) better cardinality estimations for SPARQL query graphs of a general shape, (ii) efficient heuristics of query simplifications

for large query graphs, where exact Dynamic Programming can not be applied.

- We then consider query optimization in Property Graph data model, and present the architecture of the query optimizer for Cypher query language in Chapter 5
- Finally, Chapter 6 presents our contribution to the recently proposed graph database benchmark. Namely, we describe our solution for the parameter selection problem for graph queries

CHAPTER 2

Querying Paths in RDF Graphs

Parts of this chapter were published in [39] and [35]

In this chapter we describe how an RDF triple store (RDF-3X in our case) can be extended to support two frequent types of graph queries – path and reachability queries. Namely, we take two existing algorithms – the classical Disjktra's shortest paths algorithm [23] and the recently proposed FERRARI [92] algorithm for reachability – and incorporate them into the RDF-3X engine. Our focus here is therefore not on algorithmic challenges, but on the database aspects of these two problem. In order to support new types of queries in a cost-based query engine, we need to answer the following questions:

- What are the physical operators of the query engine that implement the corresponding graph operations? To support path queries we reformulate the Dijkstra's algorithm using the merge joins of RDF-3X; for reachability queries we directly employ FERRARI index to look up reachable nodes for a given node. We also discuss further implementation details for both cases.
- How to estimate the cost of these operators in the query plan? Query optimizer chooses a specific imperative execution plan of a declarative query based on its cost. The cost, in turn, depends on the cardinality of the operators in the plan, therefore estimating the cardinality of the shortest path and reachability operators is important.

As we deal with RDF data, the datasets in this chapter are taken from the Linked Data domain (knowledge bases, biomedical data), and our solutions exploit their key properties: the corresponding graphs are very sparse and are structurally close to DAGs (albeit not exactly DAGs). Since we consider the disk-based RDF-3X system, we will assume *disk-resident* graphs throughout this chapter.

2.1. Path Query Processing in RDF-3X

We first consider extracting shortest paths that match a pattern defined by a SPARQL query. SPARQL syntax needs to be extended in order to make shortest path matching possible (Section 2.1.1). Then, we describe a simple modification of the Dijkstra's algorithm that match shortest paths on a disk-resident graph. We use this extension as a physical operator for path matching in our query engine. Finally, we incorporate path expressions in our query optimizer (Section 2.1.3), most importantly by providing a cardinality estimation method that guides logical plan construction.

2.1.1. Extension of SPARQL for Path Queries

In this section we describe the extension of the SPARQL query language that allows us to query paths in RDF database. This extension is non-standard, i.e. it is not a part of W3C recommendation (which covers reachability queries only). As we have discussed in Chapter 1, the RDF dataset can be viewed as a directed graph, where nodes correspond to subjects and objects, and edges are labeled with predicates. We define an *RDF path* of length n from a node x to a node y as an ordered sequence of triples $(\mathsf{x}, \mathsf{p}_1, \mathsf{o}_1)(\mathsf{s}_2, \mathsf{p}_2, \mathsf{o}_2) \ldots (\mathsf{s}_{n+1}, \mathsf{p}_{n+1}, \mathsf{y})$, where $\mathsf{o}_i = \mathsf{s}_{i+1}$ for $1 \leq i \leq n-1$.

Now, one can ask for the *shortest* RDF path between two given nodes x and y. In order to allow such queries in SPARQL and in tune with previous work [8, 7, 57], we enrich the SPARQL syntax with a path triple pattern. Path triple patterns resemble reguluar SPARQL triple patterns, but they contain a path variable in the predicate position. We will distinguish between regular and path variables by starting the latter with '??'. A path variable in the path triple matches the shortest path from a subject to an object in the RDF graph. For example, the path pattern

```
<Athens> ??path ?var
```

matches all the objects reachable from `<Athens>` and the shortest paths from `<Athens>` to all such objects.

Multiple triple patterns are combined in a conjunctive manner and can share regular variables, thus implying joins. We recursively define any (regular or path) triple pattern as bounded, if (i) either subject or object is constant, or (ii) it shares at least one variable with a bounded pattern. For example, the path pattern in the following group of patterns is bounded:

```
?person ??path ?city.
?city <isLocatedIn> ?country.
?country <isMemberOf> <European_Union>.
```

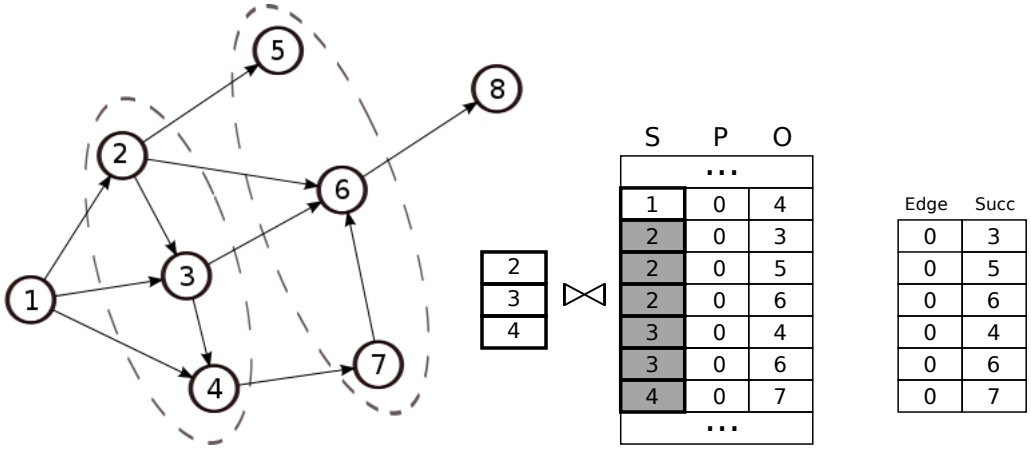
In this chapter we restrict ourselves to bounded path patterns. Informally this means that the query engine always knows the set of values of either subject or object in the path triple and thus does not have to consider all-to-all shortest paths during query execution.
Similarly to [7], we also allow specification of *filter conditions* on path variables. A filter condition can be constructed from several built-in functions, arithmetic and boolean operators. Let VR and VP be the set of regular and path variables correspondingly. We will support the following built-in function over path patterns:

- `containsAny(path, element)`, where `path` $\in VP$ and `element` $\in VR$, that checks whether `path` contains a node or edge `element`
- `containsOnly(path, element)` with the same domains, it checks whether `path` consists only of `element`. In this case `path` can be seen as sequence of zero or more `element`s.
- `length(path)` returns the length of `path`, and together with comparison operations it allows us to restrict the length of the path.

Syntactically, the path filter conditions are used in the same way as FILTER in SPARQL.

2.1.2. Join-based Dijkstra's algorithm

In order to efficiently support the language construct introduced in Section 2.1.1, we need a physical operator in our query engine that would yield shortest paths between given nodes. We build our path query processor upon the classical Dijkstra's algorithm [23]. The algorithm finds the shortest path between two nodes by traversing nodes in the graph in a breadth-first manner. Trivial modifications to Dijkstra's algorithm allow us to find the paths from the source to every other node in the graph, or to construct shortest path trees with several sources. Since the graph is disk-resident, for every visited node the algorithm needs to retrieve the list of its successors from disk. We will call this operation `getNeighbors`. A naive approach to perform `getNeighbors` for all visited nodes is to execute it independently for every node. Namely, for every visited node u we initiate a scan on the SPO index, look up the position of u in the corresponding B^+-tree, read the leaf and decompress it, and extract all triples that have u as a subject from the decompressed leaf. We can get the reverse Dijkstra's algorithm simply by scanning the OPS index, in this case at every step we get the predecessors of each processed node.



(a) Dijkstra's algorithm **(b)** Join-based `getNeighbors`

Figure 2.1.: Dijkstra's algorithm for the disk-based system

However, this multiple-lookup based implementation becomes the bottleneck of the algorithm for disk-resident graphs. Consider, for example, the graph in Figure 2.1a. The start node is 1, and we enqueue all its successors. Then, we issue three different requests to disk to get the successors of the nodes 2, 3, and 4. This results in three B^+-tree lookups. Even if all three nodes reside on the same leaf page, we will read and decompress it three times. Caching the page would solve this particular problem, but it does not scale well since the number of nodes (and pages) grows exponentially.

We can speed up this process by getting the successors of all three nodes in one database request. Namely, let us think of a pseudo-scan operator that iterates over nodes 2, 3 and 4. If we join this operator with the SPO-index scan of the whole database, we will get all the triples that start with 2, 3 and 4 respectively, i.e. exactly the successors of the nodes in consideration. Figure 2.1b gives an illustration of this idea. We join on the S-column and return values from P and O columns (labels of the edges and successors, respectively). The join (actually, the merge join) is faster than the multiple lookups due to several reasons: First, while merging, the RDF engine locates the B^+-tree leaf containing the first node, and then scans the sequence of leaves, occasionally skipping those that a priori can not contain nodes of interest. Second, if the input nodes are located on the same leaf page, we will read and decompress this page only once, as opposed to multiple extractions done by the multiple-lookup approach. Third, by combining several nodes in one request, we significantly reduce the number of total requests to the disk.

Our join-based `getNeighbors` operation takes the set of nodes as input, therefore we need to accumulate several visited nodes prior to calling it. This idea is

Algorithm 1: Join-based Dijkstra Algorithm

Input: s, d – start and destination nodes in the graph
Result: path p from s to d

```
1  begin
2      Q ← {s}
3      V_cur ← {s}
4      V_prev ← ∅
5      while Q is not empty do
6          n ← Q.dequeue
7          n.status ← processed
8          if n = d then
9              p ← reconstruct path from s to d
               ▷ requires keeping the predecessor for each node n
10             return p
11         if V_prev is empty then
12             S ← getNeighbors(V_cur)                    ▷ see Figure 2.1b
13             V_prev ← V_cur
14             V_cur ← ∅
15         V_prev ← V_prev \ {n}
16         relax nodes from S
17         foreach v ∈ S, v.status ≠ processed do
18             V_cur ← V_cur ∪ {v}
19     return [ ]                                         ▷ no path found
```

leveraged by the Join-based Dijkstra's algorithm, sketched in Algorithm 1. The algorithm alters the standard Dijkstra's algorithm in the following way:

1. We collect visited nodes in $\mathcal{V}_{cur}$ and later use them as an input for the `getNeighbors` operation. The nodes that were used for the previous call of `getNeighbors` are kept in $\mathcal{V}_{prev}$.

2. As the traversal proceeds, new nodes are added to $\mathcal{V}_{cur}$, and the scanned nodes are deleted from $\mathcal{V}_{prev}$.

3. Once $\mathcal{V}_{prev}$ becomes empty, we need to get the new portion of neighbors. The nodes that were queued in Q since the last `getNeighbors` operation (we store them in $\mathcal{V}_{cur}$) become an input for the new join-based `getNeighbors` operation call.

Consider, for example, the graph in Figure 2.1a. We start the computation with $\mathcal{V}_{cur}= \{1\}$. Since we do not have any nodes in our buffer S, we get the successors for the nodes in $\mathcal{V}_{cur}$. Nodes 2, 3 and 4 are now enqueued and added to $\mathcal{V}_{cur}$, and node 1 is removed from $\mathcal{V}_{prev}$ (indicating that for the next loop we need to load new portion of successors). Now, we request the successors for $\mathcal{V}_{cur} = \{2, 3, 4\}$ and continue the usual Dijkstra processing until we reach node 4 and delete it from $\mathcal{V}_{prev}$. After that we again request the successors for $\mathcal{V}_{cur} = \{5, 6, 7\}$, and so on. Every time the set of nodes that are processed in the `getNeighbors` is a new layer of the graph depicted with dashed lines in Figure 2.1a. Naturally, these additional operations on $\mathcal{V}_{prev}$ and $\mathcal{V}_{cur}$ can be done in $\mathcal{O}(\log n)$, thus leaving the overall complexity $\mathcal{O}(n \log n + m)$, where n and m is the number of nodes and edges, respectively. As we show in Section 2.1.4, however, the join-based Dijkstra's algorithm is an order of magnitude faster than the original one.

It is worth pointing out that the join-based technique of getting successors of several nodes can be employed by other shortest paths algorithms, including A*, *reach-* and *landmark-based* approaches [3, 32]. There are two main advantages of the Dijkstra's algorithm: (i) it finds shortest paths and not only reachable nodes, and (ii) it does not incur any preprocessing overhead. We therefore employ it as the main physical operator for path queries in RDF-3X.

In the rest of the section we refer to this physical operator as `DijkstraScan`: it takes source nodes and returns reachable nodes with paths to them. In order to get additional speed up for the `DijkstraScan`, we consider dictionary encoding tricks, described in the following subsection.

Dictionary encoding

As discussed in Chapter 1, RDF-3X assigns an integer ID to every URI and string constant, and operates on triples of integers. The ID-to-literal mapping is maintained in the global dictionary, which can be implemented as a B^+-tree or a directed mapping index [78]. In both cases, strings with close IDs reside on adjacent pages, or even within one page.

The assignment of IDs is done during the data loading on a First-Come, First-Served basis. In this case, the successors of one node can get IDs that are far away from each other. It can happen, for example, when the triples with the same subject are scattered in the input file. Recall the example in Figure 2.1a. Suppose that the graph depicted there is a subgraph of a much larger graph, such that the nodes 2, 3, 4 were assigned internal IDs 20, 50, and 100 respectively. Suppose also that every leaf of the B^+-tree SPO index contains 10 entries (every entry is a triple). In this case, entries corresponding to the 2, 3, 4 nodes are far away from each other in the SPO index, and the `getNeighbors` operation will be likely to read three different B^+-tree leaves from disk when applied to $\{2, 3, 4\}$. The problem will become worse in the next iterations, when the number of input nodes increases.

Another problem with First-Come, First-Served assignment is mapping the results of the path query execution back from IDs to strings. If the successors of the same node have IDs far away from each other, we are likely to read another page for every next node. This leads to almost random ID-to-string lookups from disk, which becomes extremely inefficient for unselective queries.

Both of these problems can be mitigated to a certain degree by assigning IDs to nodes in the breadth-first order. Initially, we generate the temporary dictionary in a First-Come, First-Served manner, so that we can operate on triples of integers.

We propose a procedure, presented in Algorithm 2, that operates on top of the temporary dictionary. It is a simple modification of the Breadth-First Traversal that starts with finding all root nodes of the graph, i.e. nodes without incoming edges. From every root node s we run a Breadth-First Search, assigning new IDs in the order of the search. If the node already has an ID, we skip it. This may happen if this node was processed during the Breadth-First Search from a previous root. Final assignment of IDs is done before the strings are loaded, and before any index is created so that created indexes already contain triples with new, Breadth-First-based IDs.

We will show in the evaluation section that `DijsktraScan` is a very efficient way to extract paths from RDF graphs. In order to fully support declarative queries with paths, however, we also need to discuss query optimization (i.e., ordering

Algorithm 2: BFS-based Dictionary mapping

Result: new mapping $\mathcal{I}[n]$ for every node in the graph

begin
  $\mathcal{I} \leftarrow \{\}$
  $\mathcal{S} \leftarrow$ set of nodes with no incoming edges
  $c \leftarrow 0$ ▷ current id
  foreach $s \in \mathcal{S}$ **do**
    $\mathcal{I}[s] \leftarrow c$
    $c \leftarrow c + 1$
    $Q \leftarrow \{s\}$
    while Q *is not empty* **do**
      $n \leftarrow Q$.dequeue
      n.status ←processed
      if $n \notin \mathcal{I}$ **then**
        $\mathcal{I}[n] \leftarrow c$
        $c \leftarrow c + 1$
      add all unprocessed neighbors of n to Q
  return $\mathcal{I}$

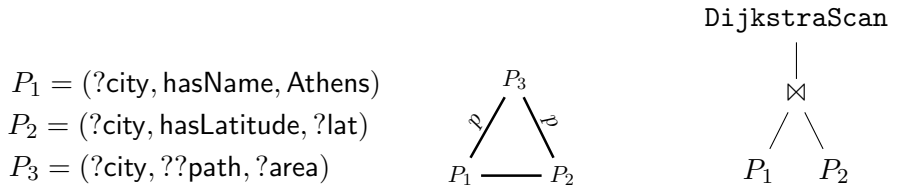
of path operators and regular joins), and we do it in the following section.

2.1.3. Query Optimization for Path Queries

As discussed in Chapter 1, the initial step in getting an optimal query plan for the query is to transform it into a calculus representation (query graph) for further optimizations. Since we extend the SPARQL syntax with the path triple pattern, we need to map this pattern onto our query graph model. We illustrate such a mapping with the following example. Consider a query that finds out the areas (country, continent) where the city Athens is located, and the city's latitude:

```
select ?city ?lat ?area ??path where {
        ?city hasName "Athens".
        ?city hasLatitude ?lat.
        ?city ??path ?area.
}
```

Each of the three triple patterns of this query (including the path triple pattern) gets its own node in the query graph, depicted in Figure 2.2a. The first two patterns (P_1 and P_2) are connected via a regular edge that denotes a join between the patterns. Additionally, they both are connected to P_3 with special p-edges.



(a) Triple Patterns and Query Graph **(b)** Join Tree

Figure 2.2.: Translating the path query into the join tree

Such edges denote the fact that bindings for the variable `?city` (coming from P_1 and P_2) will serve as an input to the `DijkstraScan` path operator, described in Section 2.1.2 (recall that since we only consider *bounded* path triple patterns, each `DijkstraScan` should have at least one such incoming edge).
If the subject or the object of the path triple is constant, then `DijkstraScan` is executed as join-based Dijkstra (reversed Dijkstra) in an asynchronous manner, like any other operator in RDF-3X. If both the subject and the object are query variables, the optimizer picks one of them depending on the expected cardinality. Then the `DijkstraScan` operator starts with computing the set of subjects (objects), and performs Dijkstra's algorithm from this set. This is the case in our example, since both `?city` and `?area` are variables, and `?city` becomes the source for `DijkstraScan`, as it is bounded by the other triple patterns. We give a plausible logical query plan in Figure 2.2b, where `DijkstraScan` correspond to the P_3 path triple pattern, i.e. it computes the bindings for `??path` and `?area` in that pattern.
As it is always the case with query optimization, the correct placement of `DijkstraScan` in the query plan (as part of join ordering) requires accurate estimation of the cardinality of corresponding path triple patterns. In this section we discuss estimators for individual path triples, which can be indexed when the data is loaded. Then, the runtime selectivity estimation merely requires one or two lookups in a small-sized B^+-tree, which is neglible compared to the actual query execution time.
We first consider the case when the path triple pattern `(s, ??p, o)` contains one constant node (subject `s` or object `o`). Then, the cardinality estimation of such a triple pattern boils down to computing the number of nodes visited by Dijkstra's algorithm starting from `s` or reversed Dijsktra's algorithm starting from `o`. The procedure estimating this number is given in Algorithm 3. We start from the nodes in RDF graph that do not have any successors ('leaves'). Then, we run breadth-first-traversal in the reversed edges direction (lines 6-11), that is, going 'up' the graph.

Algorithm 3: Cardinality Estimation for Path Queries in RDF Graphs

Result: $\mathcal{F}[n]$, $\mathcal{B}[n]$ – cardinality of BFS and reversed BFS traversals for every node in graph G

```
1  begin
2      foreach node n ∈ G do
3          F[n] ← 0
4          B[n] ← 0
5      S ← set of nodes with literal-only successors or without successors
6      while reversed Breadth-First Traversal from S do
7          foreach visited n do
8              Succ ←list of successors for n
                ▷ nodes visited before n, since this is a reversed traversal
9              foreach i ∈ Succ do
10                 F[n] ← F[n] + F[i]
11             F[n] ← F[n] + Succ.size
12     S' ← set of nodes with no predecessors
13     while Breadth-First Traversal from S' do
14         foreach visited n do
15             Pred ← list of predecessors for n
16             foreach p ∈ Pred do
17                 B[n] ← B[n] + B[p]
18             B[n] ← B[n] + Pred.size
19     return F, B
```

For every visited node, we already know the forward selectivity of all its successors since we visited them before processing the current node, so we just sum up their estimations with the number of successors and get the estimation $\mathcal{F}[n]$ (lines 9-11). We proceed with the backward estimation in the same manner (lines 12-18). The results of this procedure are materialized in the B^+-tree indexed by the constant included in the pattern.
If both subject s and object o are constant, we approximate the cardinality with $|\mathcal{F}[s] - \mathcal{B}[o]|$.
The last scenario is a triple pattern P_1 with three variables. Since we only consider bounded triples, there exists a (possibly long) join chain in the query from P_1 to a triple P_2 with a constant subject or object. If we start the Breadth-First traversal from the constant subject (object) of P_2, it will naturally reach the subject (object) of our triple. Therefore, the cardinality of P_2 can serve as an approximation of the cardinality of P_1, and this scenario is reduced to the previous one.
We measured the accuracy of Algorithm 3 by using the YAGO2 dataset and comparing the estimated cardinalities with the real cardinalities. The test workload contains 1000 random triple patterns with one constant element (subject or object), the approximation error is given by the formula:

$$\text{relative error} = \frac{\max(\textit{real cardinality}, \textit{estimation})}{\min(\textit{real cardinality}, \textit{estimation})} - 1$$

The results are shown in Table 2.1. On average (the median error) we misestimate the backward selectivity by 16% and the forward selectivity by 50%. Using an off-the-shelf laptop, it takes 5 minutes to compute the new indexes (10% of the database build time), and they increase the database size by at most 10%. On the other hand, as a result, cardinality estimation allows us to incorporate path query processing into the RDF-3X's cost model and query optimization.
The main reason why such a simple procedure suffices is that Linked Data graphs are usually quite sparse and similar to acyclic. Therefore, Breadth-First expansions from $\mathcal{S}$ (roots) and $\mathcal{S}'$ (leaves) are likely to cover most of the edges in the graph, and the counts collected during traversals accurately capture the number of reachable nodes. Note also that this procedure ignores specific edge labels along the path; if we are to take edges into accound, we have to make the independence assumption and multiply the cardinality of the path by the selectivity of the given edge predicate. We will relax this assumption in Section 2.2, where we provide an accurate estimate based on the reachability index.

Direction	min error	5% quantile	median error	95% quantile	max error	mean error
forward	0	0	0.50	4.02	70534	170.6
backward	0	0.039	0.16	4.25	20.78	0.87

Table 2.1.: Path triple selectivity estimation error for the YAGO2 dataset

	less than 1000 nodes returned		more than 1000 nodes returned	
Dictionary	join	lookup	join	lookup
new dict	4 ms	12 ms	32 ms	201 ms
old dict	18 ms	13 ms	78 ms	228 ms

Table 2.2.: Runtime of `getNeighbors` operation

2.1.4. Implementation and Evaluation

We run a set of mini-benchmark queries over two Linked Data datasets to study efficiency of our techniques. All experiments were performed on a 2-core Dell laptop with 4 Gb of memory using 64-bit Linux 2.6.35 kernel. We performed *cold cache* experiments by dropping all file-system caches and running the queries. We repeated this five times and measured the average response time for every query. For *warm cache* results we ran queries five times without dropping the caches. The SPARQL queries are given in the appendix. As the competitor for our system, we used the regular path processing in Jena [48] (GRIN [101] and BRAHMS [47] are not publicly available). We had to modify the syntax of queries to meet the syntax requirements of Jena.

For the first experiment, we use the YAGO2 dataset [45]. First, we evaluated the performance of the `getNeighbors` operation. We take 1000 random nodes, and for every node compute its successors and run `getNeighbors` with the successors as the input, thereby getting all nodes that are 2 hops away from the random seed. We compared four different approaches: the join-based algorithm vs. the multiple lookup algorithm with two types of dictionary – the traditional RDF-3X dictionary and the one proposed in this chapter. We also divide results into two groups, depending on how many nodes were returned by the operation. The obtained running times are provided in Table 2.2. It clearly identifies that the combination of the new dictionary and the join-based procedure yields the best running time for both scenarios. For the large output, this combination outperforms the naive approach (lookup with the old dictionary) by a factor of 10. However, if the number of nodes obtained from the disk is relatively small, the difference between different techniques is not that large due to the join execution overhead. We also measured the runtimes of the Dijkstra's algorithm in the same four scenarios. Table 2.3 gives evidence that the proposed combination

	less than 1000 nodes returned		more than 1000 nodes returned	
Dictionary	join	lookup	join	lookup
new dict	5 ms	8 ms	120 ms	1052 ms
old dict	12 ms	8 ms	458 ms	1169 ms

Table 2.3.: Runtime of the Dijkstra's algorithm

Dataset	Query	RDF-3X		Jena	
		cold cache (warm cache)			
YAGO2	Q1	0.18	(0.004)	0.45	(0.10)
	Q2	1.09	(0.19)	34.54	(1.31)
	Q3	1.21	(0.09)	28.17	(1.07)
	Q4	1.12	(0.18)	29.98	(0.95)
	Q5	2.49	(1.39)	> 30 min	(> 30 min)
	geom. mean	0.92	(0.11)	> 29.83	(> 2.99)
UniProt	Q1	0.52	(0.01)		
	Q2	4.01	(3.21)	> 30 min	(> 30 min)
	Q3	11.54	(4.61)		
	geom. mean	2.88	(0.53)	> 30 min	(> 30 min)

Table 2.4.: Query runtimes in seconds for YAGO2 and UniProt datasets

of the clustered dictionary and the join-based `getNeighbors` yields the best performance, outperforming the rest by an order of magnitude when the number of visited nodes is large.

Then, we ran 5 different queries starting from the *geographical hierarchy of Ulm* (Q1) to *all people from Germany that died somewhere in France* (Q4) and *all Mediterranean-born European scientists that are known for some physical phenomenon* (Q5). The full text of the queries can be found in Appendix A. In general, the performance of Jena significantly degrades as the number of joins in the query increases. As reflected in Table 2.4, our approach outperforms Jena by a large margin, improving cold cache times by a factor of 32, and warm cache times by a factor of 27 in the geometric mean. RDF-3X was consistently better for all queries, gaining a factor of 1400 speed-up for the last query.

To quantify how our individual techniques affect performance, we ran the experiments with the variants where *only join-based Dijkstra* and *only new dictionary* are used. The results are also presented in Table 2.5. As we see, both techniques play a significant role in ensuring high performance, gaining the speedup of a factor of 4.7 and 3 correspondingly in the *cold cache* case.

For the second experiment, we use the UniProt dataset [102]. We ran three

Technique	Q1	Q2	Q3	Q4	Q5	geom mean
			cold cache			
baseline	0.18	1.09	1.21	1.12	2.49	0.92
only new dict	0.69	4.85	5.20	8.63	11.03	4.40
only join-based Dijsktra	0.18	7.71	3.82	3.86	8.64	2.81
			warm cache			
baseline	0.004	0.19	0.09	0.18	1.39	0.11
only new dict	0.04	0.99	0.75	0.54	5.87	0.62
only join-based Dijsktra	0.004	0.41	0.29	0.36	4.86	0.24

Table 2.5.: Effect of Individual Techniques for Path Processing, YAGO2 dataset

different queries with increasing number of joins in them. The full text of the queries can be found in Appendix A. The results are given in Table 2.4. Jena also performs very poorly in this scenario, in fact, it was not able to evaluate any of the queries in 30 min., as opposed to the average response time of 2.88 seconds of RDF-3X.

2.2. Reachability Queries in RDF-3X

While the path queries described in the previous section are not part of the SPARQL standard, the recent draft of SPARQL 1.1 offers a "close enough" replacement of them, namely *property path* expressions, which can be seen simply as regular expressions over paths between two nodes in the RDF graph. These expressions do not allow to query paths themselves, but rather to impose connectivity conditions with paths consisting of given edge labels. In this section we will describe efficient mechanisms to support reachability queries in RDF-3X.

2.2.1. Expressing reachability queries in SPARQL 1.1

The simplest SPARQL query with a regular path expression has the form

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

and retrieves the nodes `?s` and `?o` connected via the path that matches a regular expression `path`. In this work we consider regular expressions over constant predicates with disjunction (denoted by '|'), path concatenation ('/') and Kleene star ('*') (corresponding to zero or more occurrences of a predicate) and its variant '+' (one or more occurrences). The expression specifies a sequence of predicates along the path from `?s` to `?o` in the RDF graph. We call a triple pattern `(?s, path, ?o)` with regular path expression a *regular path* pattern.

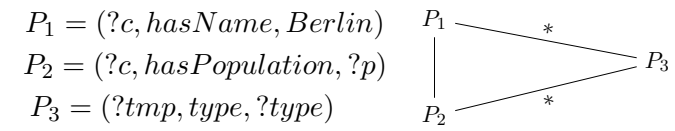
As an example, the following query retrieves the entities that can be reached from Berlin by the path consisting of zero or more predicates `isLocatedIn` and then gets the types of these entities. In other words, it looks up the geographical hierarchy of Berlin (Berlin, Germany, Europe, Earth) and returns the types of the corresponding objects (i.e., a state, a member of EU, a continent, a planet etc.):

```
select * where {
        <Berlin> isLocatedIn*/type ?type.
}
```

The early version of the W3C SPARQL standard allowed asking how many paths between two nodes matching the given regular expression exist. However, the part of the W3C standard defining counting property paths was demonstrated to be computationally intractable. It has been shown [9, 10, 66] that the problem lies in using bag semantics for path queries, that is, in counting all the different paths that can reach a given node via a specified sequence of predicates. Counting paths leads to returning multiple copies of the same node if it is reachable by several distinct paths. This results in a double exponential lower bound on the result set size of a single query even for very small graphs and simple regular expressions like `predicate`$*$, rendering the original W3C semantics infeasible. It is also known [9] that even restricting ourselves to simple paths (without repeating nodes) does not make counting easier, neither does the acyclicity of the underlying graph help.
For these reasons, the current W3C standard suggests (and our implementation supports) an intuitive existential semantics that merely checks if there exists a specified path between two nodes, without counting the number of such paths. In other words, we treat the regular path pattern `(?s, path, ?o)` as a *reachability query* that returns all pairs of nodes reachable by the given path. Naturally, `?s` and `?o` may appear in other triple patterns as well, thus restricting us to subjects with specific properties that can reach objects with other properties (of course, these properties can be expressed with arbitrary complex SPARQL subqueries).

2.2.2. Reachability Query Planning

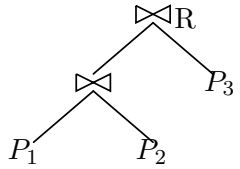
As with path queries (Section 2.1), we first need to define a translation of SPARQL with property paths into our query graph model. We map new regular path triples onto query graph nodes and edges in the following way. First, if the regular expression on the path contains a sequence of steps (i. e., the path concatenation '/' is used), we replace such a triple pattern with a sequence of patterns, introducing temporary variables. This way, the pattern `?s`



(a) Triple Patterns and Query Graph

$P_1 \bowtie P_2$
$P_1 \bowtie_R P_3$
$P_2 \bowtie_R P_3$

(b) Potential Joins



(c) Join Tree

Figure 2.3.: Translating the reachability query into the join tree

`isLocatedIn*/type ?o` expands into two patterns `?s isLocatedIn* ?tmp . ?tmp type ?o`, where the second pattern does not require any path processing. From now on, we can assume that every regular path expression simply requires matching a single predicate zero or more times (or one or more types for the '+' operator). Then, every such path triple pattern is expressing the reachability requirement on its subject and object, that is, the subject should reach the object via the given path. Such a requirement is encoded as a special *reachability edge* between all the patterns containing subject and object of the given path reachability triple.

```
select * where {
        ?city hasName "Berlin".
        ?city hasPopulation ?p.
        ?city isLocatedIn+/type ?type.
}
```

Query 2.1: Reachability query retrieving the geographical hierarchy and population of Berlin

For example, Query 2.1 will have its third triple pattern rewritten into

```
?city isLocatedIn+ ?tmp.  ?tmp type ?type
```

The corresponding query graph, depicted in Figure 2.3a, therefore has three nodes. The edge between P_1 and P_2 expresses the fact that these two patterns share the variable `?c`, while two reachability edges (depicted with *-label) mean that there should exist a path between `?c` defined in P_1 and P_2 and `?tmp` in P_3.

Nodes and edges in the query graph correspond to logical operators in the query plan. As usual, nodes are mapped to scans over the entire database (or the corresponding index) with selections induced by literals. Edges are transformed into joins, such that regular edges become equi-joins and reachability edges turn into *reachability joins*. A reachability join $\bowtie_R (?s, ?o)$ is conceptually a join with the condition `?s` *reaches* `?o` as a join predicate (as opposed to `?s=?o` condition in an equi-join). For Query 2.1, the two reachability joins and an equi-join are given in Figure 2.3b. Out of them, the join tree in Figure 2.3c is constructed. Note that since the query graph is a clique, only two of the three potential joins are used. Indeed, the equi-join between P_1 and P_2 and the reachability join between their result and P_3 guarantees that all three join conditions are satisfied.
There are two special cases of reachability joins:

- Either subject or object in the corresponding reachability triple pattern is constant (e.g., in the triple `Berlin locatedIn* ?place`). Then the reachability join $\bowtie_R (?s, ?o)$ turns into a reachability selection $\sigma_R(const, ?o)$.
- Either subject or object is unbound, i.e. it does not occur in any other triple pattern. In this case $\bowtie_R$ becomes a reachability scan: for a bounded variable it looks up all the reachable nodes via the specified path.

After we interpreted the query graph in terms of table scans, projections and joins, we can run one of the classical join ordering algorithms to obtain the optimal query plan. To be able to incorporate the new reachability join into Dynamic Programming, we discuss the algebraic properties of the operator first.

Algebraic Properties of $\bowtie_R$

First, we need to refine our notation of the reachability join operator. Let $\mathcal{F}(P)$ be the set of variables of an expression P. This expression can be a simple triple pattern, or a result of an operator (e.g. a join) on several other expressions. The reachability operator $P_1 \bowtie_{R(v \to w, p)} P_2$ is defined for expressions P_1 and P_2, where $v \in \mathcal{F}(P_1)$ and $w \in \mathcal{F}(P_2)$. It returns all possible combinations of bindings of variables in P_1 and P_2 (just like an equijoin), under the condition that v reaches w via the path confirming to an expression p. For brevity, we will omit the regular path expression p, since all the properties do not depend on a specific regular expression.
Unlike equijoin, however, the reachability join has a fixed order of operands (v has to reach w, not the other way round), since reachability in directed graphs is not symmetric. We can not, therefore, talk about commutativity of $\bowtie_{R(v \to w)}$, since reordering of operands can be undefined.

The operator $\bowtie_R$ is associative due to transitivity of reachability in directed graphs:

$$(P_1 \bowtie_{R(v \to w)} P_2) \bowtie_{R(w \to t)} P_3 = P_1 \bowtie_{R(v \to w)} (P_2 \bowtie_{R(w \to t)} P_3),$$

where $v \in \mathcal{F}(P_1), w \in \mathcal{F}(P_2), t \in \mathcal{F}(P_3)$
Additionally, it is associative with regards to the equijoin operator:

$$(P_1 \bowtie P_2) \bowtie_{R(v \to w)} P_3 = P_1 \bowtie (P_2 \bowtie_{R(v \to w)} P_3),$$

with $v \in \mathcal{F}(P_2)$ and $w \in \mathcal{F}(P_3)$
Associativity allows us to transparently incorporate $\bowtie_R$ into the Dynamic Programming algorithm of RDF-3X given in Chapter 1.

2.2.3. Reachability Operator

The physical operator that executes $\bowtie_R$ is based on a reachability index; essentially, for every pair of nodes such an index yields whether they are connected or not via a path that confirms to a specified regular expression. In this section we describe a state-of-the-art index FERRARI [92] that we used to support $\bowtie_R$ in RDF-3X. The index is constructed while bulk loading the data into the database system.
To simplify the exposition, we assume that all edges in the graph have the same label. We will later lift this artificial assumption and describe the reachability index construction for graphs with multiple edge labels.
The first step in constructing the reachability index is to compute the Strongly Connected Components: indeed, every node in one SCC can reach any other node in this SCC. After that, we keep the mapping $node \to SCC$, which allows us to give a positive answer to reachability question for nodes from the same SCC. This mapping essentially merges all nodes from the same SCC into one meta-node, so that the resulting graph is directed and acyclic (DAG). It is this DAG that becomes an input for the reachability index construction.
FERRARI index is based on the classical reachability index by Agrawal et al. [2] that assigns postorder numeric IDs to nodes of the graph as a result of the depth-first traversal. Let $\pi(v)$ denote this postorder ID of a node v, then the set of nodes reachable from v in a subtree T rooted at v can be captured by a single interval [92]:

$$I_T(v) = \left[\min_{w \in T} \pi(w), \pi(v)\right]$$

We can therefore answer whether a node w is reachable from v *in this tree* T by simply testing the containment of $\pi(w)$ in the interval $I_T(v)$. A single interval

would only bring us reachability queries in the tree-like structures, so for DAGs we need to store multiple intervals (as a set $\mathcal{I}(v)$) to cover potential non-tree edges. Specifically, Agrawal et al. [2] showed that it suffices to visit vertices of the graph in reverse topological order, and for the current vertex v, the intervals of its children are added into the interval set $\mathcal{I}(v)$. Reachability queries are then resolved by checking whether a postorder ID $\pi(w)$ is contained in *any* of the intervals of the set $\mathcal{I}(v)$.
Since this classical interval-based reachability index is in fact a representation of the transitive closure of the graph, it has the worst-case complexity of $\mathcal{O}(n^3)$ (both construction and space consumption). The main idea behind the FERRARI index then is to merge some of the adjacent intervals in $\mathcal{I}(v)$ to create a more compact representation. By merging we allow some false-positive results which correspond to the "gaps" between original intervals that got merged. Some of the intervals in $\mathcal{I}(v)$ are kept as *exact* (subject to available space budget). During runtime of the reachability query, there are three possibilities:

- if the target id is contained in an exact interval of the source node, yield a positive answer
- if the target id is outside of any interval of the source, yield a negative answer
- if the target id is in one of the approximate intervals of the source, start an online search (a depth-first traversal with heuristics) to determine actual reachability

The reachability index described above does not take into account different predicate labels. Therefore, for every distinct predicate in the RDF dataset, we create a separate index for the subgraph induced by edges with that predicate label. In order to reduce the overhead, we only consider predicates that label both incoming and outgoing edges from the same node, since these are the only predicates that can form paths in the graph. In reality, the YAGO2 dataset (over 100 million triples) contains 15 such predicates, the sizes of the corresponding induced subgraphs vary from few hundred nodes to 5 million nodes, and the total indexing time amounts to roughly 90 seconds on an off-the-shelf laptop computer. The total size of all FERRARI indexes for YAGO2 amounts to 210 MB (ca. 3% of the total database size), permitting to maintain them in main memory.

Cardinality Estimation using FERRARI

In order to enable the cost-based query optimization, we need to extend the cost model to account for reachability joins and selections. More specifically, the

optimizer has to estimate the result sizes (cardinalities) of reachability selections and reachability joins. To estimate the cardinality of $\sigma_R(const, ?o)$ recall that the FERRARI index for the *const* element contains sequence of exact and approximate reachability intervals corresponding to the set of nodes reachable from a particular vertex. The intervals are approximate in the sense that all reachable nodes are covered (i. e., nodes outside the intervals are not reachable from *const*), while they admit false positive answers to the reachability query. The total size of these intervals therefore serves as a fast and accurate approximation for the cardinality of the $\sigma_R(const, ?o)$ operator. In practice, a large fraction of predicates in Linked Data graphs do not form cyclic paths, because they express hierarchical dependencies (like `childOf`, `locatedIn`, `subClassOf`). In these cases the labeling intervals in FERRARI are always exact, i.e. they correspond exactly to the nodes reachable from *const*.
In order to estimate the cardinality of $\bowtie_R (?s, ?o)$, for every predicate we precompute and materialize the average number of nodes contained in the set of intervals of a vertex in the FERRARI index, i.e. the total size of intervals for all nodes divided by the number of nodes. This value provides a rough approximation of the number of nodes reachable from the fixed node `?s`. Then, to estimate the result size of $\bowtie_R (?s, ?o)$ we multiply this number by the cardinality of the subplan yielding the `?s` values (that is, by the expected number of start nodes).

2.2.4. Runtime Techniques

Once the optimal logical plan has been found, the physical operators are constructed. In the pipeline model, the reachability join operator is implemented similarly to a hash join: in the *build part*, the more selective input subplan is executed, and then in the *probe part* the output of the second subplan is checked against the build part and FERRARI index, to find the nodes that are reachable from any of the nodes in the build part. The reachability scan and reachability filter simply probe the FERRARI index for every incoming node to filter those that satisfy reachability constraints.
Even after the optimal query plan has been identified by the system, its execution suffers from the fact that individual triple patterns may be surprisingly unselective, while their combinations (i. e., joins) rule out most of the scanned data. In order to deal with these phenomena, RDF-3X employs the Sideways Information Passing (SIP) mechanism [77]. The system keeps the information about significant gaps occurred in scans and merges and about domains of hash joins. All this information is passed between different index scans via shared-memory variables.
Unlike merge joins and index scans, our reachability join operator does not keep

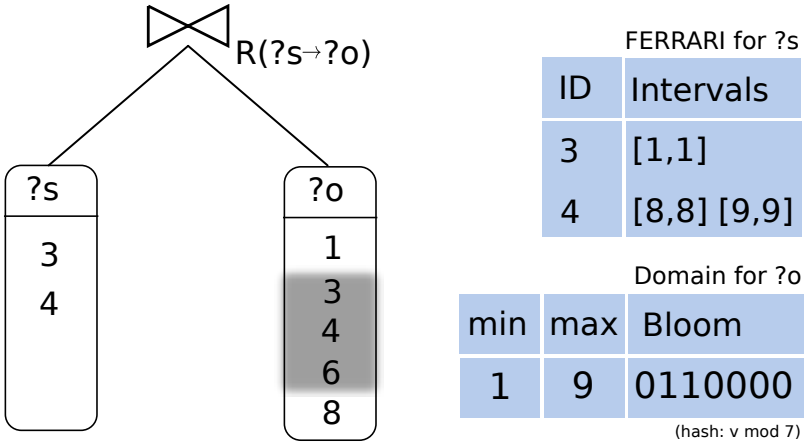


Figure 2.4.: Sideways Information Passing for reachability joins

the order of the input join variables values, so we can not identify potential gaps in its output and pass it to the next operators. However, we can still leverage the SIP mechanism to speed up the reachability join itself. Recall that the intervals of the FERRARI index contain all the reachable nodes for all the `?s` values of the left side. By keeping these reachability intervals in the shared memory for the `?o` variable of the right side, we can notify the right side about the potential gaps in the values of `?o`. Intuitively, during the build time we identify for every `?s` value the nodes that can not be reached from `?s`, thus allowing to skip these values in the right side subplan execution.

Instead of keeping in the shared memory the intervals themselves, we encode these potentially reachable nodes in a Bloom filter. As the Bloom filter will be probed for absence of certain ranges of values, we use the range-preserving hash function of the form $h(x) = ax \mod m$ [77]. We also keep the minimum and maximum values of potentially reachable nodes, since they can also guide the underlying index scans of the right side to skip some fractions of the data.

The SIP mechanism for the reachability join is illustrated in Figure 2.4. Suppose that the inputs to the reachability join are two index scans, and during the probe time we identify that the left scan yields the values 3 and 4. Then, by looking up the corresponding intervals in the reachability index, we figure that the potential domain for `?o` variable has values 1, 8, 9. We encode these values in a Bloom filter, and now during the right scan we know that we can skip all values between 1 and 8 (not inclusive, showed in gray in Figure 2.4), potentially also skipping several disk pages between these values.

As a concrete example, consider again Query 2.1 and its optimal plan in Figure 2.3c. The left side (build part) of the reachability join gets a very selective subplan (effectively, its a single tuple lookup), whereas the right side (probe

part) entails a very expensive index scan that touches a large portion of the database. The SIP mechanism provides the right side the hints to skip the most part of the scanned data, thus significantly improving the performance: on a commodity laptop the running time has decreased from 2193 ms to just 75 ms.

2.2.5. Evaluation

In our experiments we formulate queries against YAGO2 [45]. The full text of the queries is given in Appendix B. We run the modified RDF-3X system on a dual-core laptop equipped with 4 GB of main memory using 64-bit Linux (2.6.35 kernel). For comparison, we take the commercial open-source Virtuoso 7 system. The Virtuoso system is run on a server with two quad-core Intel Xeon CPUs (2.93GHz) and with 64 GB of main memory using Redhat Enterprise Linux 5.4. Table 2.6 reports the warm cache results of seven queries against YAGO2. As we see, our approach (RDF-3X with SIP) outperforms Virtuoso by a large margin, providing the runtime that is 3-5 times faster than the competitor's. Virtuoso could not execute the last query, the query execution process has reported "out of memory" exception. Note also that the Virtuoso instance has allocated 20 Gb of main memory, whereas RDF-3X used less around 1 Gb of main memory for query execution. We additionally quantify the effect of Sideways Information Passing, by switching it off and running the same queries in RDF-3X. As the last row of Table 2.6 shows, the impact of SIP is very significant, reaching an order of magnitude improvement for Query 4. Note that the biggest effect of SIP is achieved on the queries where the difference between cardinalities of the probe and the build sides of the reachability join is the largest, so SIP helps skipping most of data on the probe side (e.g., just a few tuples on the probe and a large `IndexScan` on the build). This happens, for instance, when we traverse geographical hierarchies of a specific place (small probe side), and return certain properties (names) of all traversed objects (large right side).

	Q1	Q2	Q3	Q4	Q5	Q6
RDF-3X (with SIP)	1	188	1	75	350	253
Virtuoso 7	8	452	4	418	946	–
RDF-3X (w/o SIP)	13	973	14	2193	401	560

Table 2.6.: Reachability queries in RDF-3X (with and without Sideways Information Passing) and Virtuoso 7, ms

2.3. Related Work

Algorithms for reachability and shortest path problems have been intensively studied in the database community; a survey of this research can be found in [1]. The problem that we pursue in this chapter is orthogonal to this line of work. We take the classical Dijkstra's shortest path algorithm and the state-of-the-art reachability index FERRARI [92], and build a full-fledged path and reachability query processing system on it. Besides, we propose low level database techniques which speed up Dijkstra's algorithm on disk-resident graphs, that can be applied to other shortest-path algorithms as well.

Our cardinality estimation problem is related to counting different structures (paths, subgraphs, simple and Hamiltonian cycles) in graphs — a well-developed field in discrete mathematics [6, 5, 85, 26]. Query optimization, however, imposes strict requirements to the runtime of the estimator. In particular, the nature of the cardinality estimation problem requires that the solution be of an almost constant runtime complexity: during the compilation time of a query the optimizer needs to estimate the cardinality of every subquery in the exponentially growing search space. This forces us to develop robust heuristics that are simpler and more robust than existing ones, but probably do not have any worst-case error bounds on random graphs.

The languages for path queries over graph-structured data have been the subject of much investigation in the theoretical community, see for example the survey done by Barcelo et al. [12]. The focus there, however, is on expressivity and complexity of the query language, and not on its efficient implementation.

Several frameworks and prototypes for RDF path queries have been described in the literature. Namely, Anyanwu et al. [7] propose a path query evaluation framework that relies on expensive matrix decomposition for the precomputation step ($\mathcal{O}(n^3)$, where n is the number of nodes in the RDF graph) and therefore cannot scale to large graphs. The GRIN engine [101] concentrates on providing an index for graph queries utilizing graph partitioning, but the construction time for such an index is also prohibitively long ($\mathcal{O}(n^4 \log n)$). BRAHMS [47] is another example of an engine that supports graph traversal queries. However, it only finds the paths with a predefined (fixed) length. DOGMA [18] is a disk-based graph pattern matching index, but it does not support path query processing.

Finally, adoption of SPARQL 1.1 standard has led commercial systems like Virtuoso to support reachability queries, in a manner similar to recursive SQL support. We have shown that specialized algorithms and indexes significantly outperform such an approach.

CHAPTER 3

Approximate Paths in Small-World Graphs

Parts of this chapter were published in [40] and [38]

In the previous chapter we discussed the Dijkstra's algorithm over RDF graphs such as YAGO2. Unfortunately, the algorithm's performance degrades quickly as the average node degree in the graph increases: indeed, as we show in this chapter, the queue of nodes used in the algorithm frequently contains most of the graph's nodes when run against a dense small-world graph. Therefore, an application that needs to compute shortest paths over large dense graphs, has to rely on heuristics. An example scenario may include: (i) social network analysis, where computing shortest paths is a building block for centrality computation and other analytical algorithms, (ii) interactive applications such as *How You're Connected* feature of LinkedIn that shows a path between two persons in the network.

- We first consider the shortest path estimation problem, i.e. finding a path between two given nodes that is as close to the shortest path as possible (we discuss the notion of "closeness" for paths in Section 3.1). We take the classical *distance oracle* by Thorup and Zwick [97] (or Sketch [22]) and augment it with paths between the nodes in the oracle. The goal is to precompute and store an $\mathcal{O}(n)$ sized *sketch* of the graph so that any shortest path query can be answered approximately but with high accuracy.
- We then turn to a general problem of connecting k nodes in the graph such that an induced subgraph has the smallest size (the *Steiner tree problem*). We show that our augmented Sketch index can be used to efficiently approximate Steiner trees in a procedure that combines the index use with limited local search on the original graph.

In both cases we have implemented our algorithms inside the RDF-3X engine, and compared them to the state-of-the-art approaches.

3.1. Shortest Path Estimation

3.1.1. Problem Difficulty

What makes the shortest path computation particularly hard on large graphs? Dijkstra's algorithm, the classical technique to compute the shortest path between two nodes in a graph has the asymptotic runtime complexity of $\mathcal{O}\big(m + n \log n\big)$, where n is the number of nodes and m is the number of edges [21]. On one of the benchmark datasets that we use in this chapter – the Orkut social network comprising about 3 million nodes and 220 million edges – a straightforward implementation of Dijkstra's algorithm takes more than 500 seconds on average. The reason for this is that Dijkstra's algorithm has to construct and maintain shortest paths to all nodes in the graph whose distance to the source node is smaller than the distance from the source node to the destination node. Consequently, the memory consumption of Dijkstra's algorithm is very high, requiring to maintain a number of 2.5 million nodes in the heap for the Orkut dataset, which is prohibitively expensive for simultaneous execution of many queries.

The naïve alternative of precomputing all-pairs shortest paths distances and maintaining them on disk for quick lookups is practically infeasible, requiring $\mathcal{O}\big(n^2\big)$ space and the time proportional to the output size [91]. For general graphs, it has been shown that constant query time for exact shortest path distance queries is only achievable with super-linear space requirements during preprocessing [97]. Relaxing the exactness requirement, a number of *distance oracles* have appeared which aim at providing a highly accurate estimate of the node distances [20, 89, 93, 97, 106]. The main application for these approaches occur in *geographic information systems* (GIS) or, specifically, in route planning over transportation networks. Techniques developed in this domain exploit in a crucial way special properties of transportation networks such as their near planarity, low node-degree and the presence of a hierarchy based on the importance of roads [13, 14]. These properties also help in devising faster variants of Dijkstra's online algorithm by combining it with A*-search and other goal-oriented pruning strategies [31].

On the other hand, the social networks that we consider in this chapter do not exhibit the same properties as road networks. It is well known that social networks contain many high degree nodes, are nowhere close to planar, and typically have no hierarchical structures that can be exploited for improving shortest path queries. Potamias et al. [80] make the important observation that even a 2-approximation, which is considered highly competitive for general graphs, is insufficient in the case of social networks as the distances are already

very small. Further, none of the prior work has explicitly addressed the problem of providing the shortest paths themselves, not just the node distances, as we do in this work.

We also briefly mention here that database research has considered queries over different forms of graph databases apart from road networks, such as XML graphs and biological networks [49, 50, 90, 99]. The focus, however, has been primarily on answering *reachability queries*, not in computing the shortest path distance or the shortest path itself.

Contributions. We build upon the recently proposed sketch-based framework [22], which in turn is based on the classical landmark-based approach used in distance oracles [97]. The goal is to precompute and store a $\mathcal{O}(n)$-sized *sketch* of the graph so that any distance query can be answered approximately but with high accuracy. The prior work considered maintaining only the distance information as part of the sketch. However, we observe that for social network graphs (and actually for any real-world graph data apart from road networks), the path lengths are small enough to be considered almost constant [72], and thus propose to store the complete path information (i. e. the information about constituent nodes and edges) as part of the sketch. Based on the availability of such *path-sketches* (as opposed to the distance-sketches in [22]) we develop a set of lightweight algorithms that can approximate shortest paths between any two nodes with at most 1% error. Additionally, we can also generate a set of paths (i. e. the identity of nodes in the path) that correspond to the estimated shortest path distance with no additional overhead – a feature hitherto not available.

Organization. The remainder of this section is organized as follows: First we explain the previously proposed distance oracle algorithm devised by Das Sarma et al. [22] (in the following referred to as the *sketch* algorithm) and show how simple yet powerful modifications to it yield shortest path estimates of higher quality. Subsequently, we describe a new algorithm that uses the data obtained in the precomputation step of the sketch algorithm and returns paths whose accuracy beats the previous approaches by an order of magnitude at the expense of only a marginal time overhead. In Section 3.1.5 we describe the implementation details of our algorithms within RDF-3X. Section 3.1.6 comprises a comprehensive experimental evaluation that shows the practicability of our algorithms both in terms of query response time as well as in the approximation quality of the returned results.

3.1.2. Sketch Algorithm

We start with providing a concise introduction to the algorithm devised by Das Sarma et al. [22] and its modified version, which lays the groundwork for our

approach.

Preliminaries

Let $G = (V, E)$ denote a directed graph with vertex set V and edge set E.

Paths and Distances. A path p of length $l \in \mathbb{N}$ in the graph is an ordered sequence of $l+1$ vertices, such that there exists, for every vertex in the sequence, an edge to its subsequent vertex, except the last one:

$$p = (v_1, v_2, \ldots, v_{l+1}) \quad \text{with} \quad v_i \in V, 1 \leq i \leq l+1, \tag{3.1}$$

$$(v_i, v_{i+1}) \in E,\ 1 \leq i < l. \tag{3.2}$$

For a node $v \in V$ of the graph we denote by $\mathcal{S}(v)$ the set of the successors of v in G, that is the set of vertices $w \in V$ with $(v, w) \in E$. Thus, we can express requirement (3.2) equivalently as $v_{i+1} \in \mathcal{S}(v_i),\ 1 \leq i < l$.
We write $|p| = l$ to denote the length of the path p. For vertices $u, v \in V$, let $\mathcal{P}(u, v)$ be the set of all paths that start in u and end in v. The distance from u to v, denoted by $\texttt{dist}(u, v)$, is the number of edges in the shortest such path – or infinity if v is not reachable from u:

$$\texttt{dist}(u, v) := \begin{cases} \arg\min_{p \in \mathcal{P}(u,v)} |p| & \text{if } \mathcal{P}(u, v) \neq \emptyset, \\ \infty & \text{else.} \end{cases} \tag{3.3}$$

Path Approximation. Given two vertices $u, v \in V$, let p denote a shortest path (note that there could be many) from u to v, that is, a path starting in u and ending in v with length $|p| = \texttt{dist}(u, v)$. Furthermore, let q be an arbitrary path from u to v. By regarding q as an approximation of the shortest path p, we can define the approximation error of this path as

$$\texttt{error}(q) := \frac{|q| - |p|}{|p|} = \frac{|q| - \texttt{dist}(u, v)}{\texttt{dist}(u, v)} \in [0, \infty]. \tag{3.4}$$

Path Concatenation. Let $p = (u_1, u_2, \ldots, u_{l_1}, u_{l_1+1})$ and $q = (v_1, v_2, \ldots, v_{l_2+1})$ denote paths of lenghts l_1 and l_2 respectively. Suppose $u_{l_1+1} = v_1$, that is, the last node in path p equals the first node in path q. Then, we can create new path, denoted by $p \circ q$, of length $l_1 + l_2$ by concatenating the paths p and q:

$$\begin{aligned} p \circ q &= (u_1, u_2, \ldots, u_{l_1}, u_{l_1+1}) \circ (v_1, v_2, \ldots, v_{l_2+1}) \\ &:= (u_1, \ldots, u_{l_1}, v_1, \ldots, v_{l-2+1}). \end{aligned} \tag{3.5}$$

Sketch Algorithm

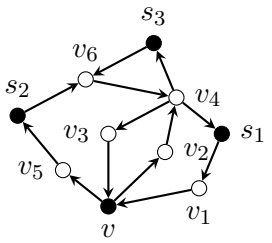
The sketch algorithm [22] approximates the shortest path distance between two given nodes in general graphs using a landmark-based approach. In order to answer a distance query for a pair of nodes (s, d) in real time, the algorithm employs a two-staged approach: a precomputation step to generate sketches (distances from all vertices to so-called landmark nodes) beforehand, and an approximation step that uses this precomputed data to provide a very fast approximation of the node distance at query time. It works by combining the two distances $\texttt{dist}(s, l)$, $\texttt{dist}(l, d)$ of the query nodes to/from a selected landmark node l into the approximated distance

$$\tilde{d}(s, d) \leq \texttt{dist}(s, l) + \texttt{dist}(l, d) .$$

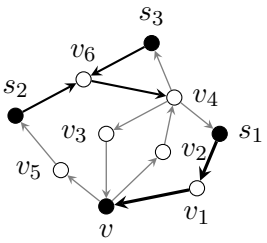
Therefore, in the original paper [22], the authors suggest to store for every node v the distances $\texttt{dist}(v, \cdot)$ and $\texttt{dist}(\cdot, v)$ from (to) the node to (from) certain landmark nodes as the result of the precomputation. This set of node-landmark distances is called *sketch* of a node.
Instead of keeping just the distances, we modify the precomputation step to store the distances along with the actual paths. Since we consider small-world graphs, the paths are expected to be relatively short. Therefore, the storage overhead of maintaining full path information as part of the sketch is not substantial – our experiments show that it is no more than twice the sketch with only distance information. Obviously, we do not incur any additional computational overhead during the precomputation step, since we require no more information than generated by the breadth-first search and reverse breadth-first search steps of sketch computation.
Also note that while the original algorithm returns an estimate of the distance between the query nodes, our algorithm returns more than just one approximate path between the query nodes, namely a queue of such paths (sorted in ascending order by path length). By providing many candidate paths, this modification could prove useful in scenarios where – for example – constraints on certain nodes/edges must be satisfied.
In the following subsections we explain these two building blocks of the (modified) sketch algorithm in detail.

Sketch: Precomputation

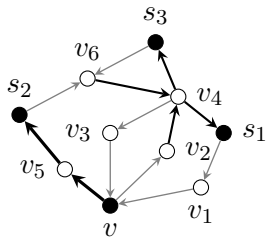
The precomputation step involves sampling some sets of nodes, computing for every node in the graph a shortest path to and from a member of this set and storing the thus obtained set of paths on external memory. These paths



(a) Vertex v and seed set $S = \{s_1, s_2, s_3\}$



(b) BFS from S in forward direction reaches v



(c) BFS from S in backward direction reaches v

Direction	Landmark	Path
$\leftarrow$	s_1	(s_1, v_1, v)
$\rightarrow$	s_2	(v, v_5, s_2)
$\vdots$	$\vdots$	$\vdots$

(d) Resulting entries in Sketch(v)

Figure 3.1.: Sketch precomputation example

will be used in the approximation step later. The preprocessing, illustrated in Figure 3.1, works as follows:

1. **Seed Set Sampling**
   Let $r := \lfloor \log(n) \rfloor$ where $n = |V|$. We uniformly sample $r + 1$ sets of nodes (called seed sets) of sizes $1, 2, 2^2, \ldots, 2^r$ respectively. The selected sets are denoted by $S_0, S_1, \ldots, S_r$.

2. **Shortest Path Computation**
   For each of the sampled seed sets S_i and every node $v \in V$ we compute a shortest path $p_{S_i \to v}$ that connects any member of the seed set to v (note that there could be more than one such path). We use breadth-first expansion from S_i and build the complete shortest path tree. For every node v, we thus obtain the closest seed node, denoted by l_1. Likewise, we compute a shortest path $p_{v \to S_i}$ that connects v to the seed set, using breadth-first expansion from S_i on reversed edges, terminating at the first seed node, denoted by l_2. The nodes l_1, l_2 are called *landmarks* of S_i for the vertex v.

This precomputation routine – that is, seed set sampling and shortest path computation – is repeated k times, thereby generating for each vertex $v \in V$ a number of $2rk$ landmarks and paths (at costs of the same number of BFS expansions from the seed sets). Note that the set of selected landmarks might be different for every node. The data (sketch) gathered for node v, consisting of the $2rk$ landmarks and paths, is denoted by $\mathsf{Sketch}(v)$. As a result of the precomputation step, we store the sketches of all nodes on disk.

Sketch: Shortest Path Approximation

In the second stage, the algorithm receives a pair of nodes, (s, d), as input. The goal is to compute, in real time, a path $p_{s \to d}$ from s to d that provides a good approximation of the shortest path, that is, a path with small $\texttt{error}(p_{s \to d})$. The sketch algorithm, presented in Algorithm 4, performs the following steps to generate such a path:

1. Load the sketches of nodes s and d from disk
2. Let L be the set of common landmarks in the sketches. Note that for undirected graphs we can guarantee (for nodes in the same component) that there exists at least one such landmark, because the sketches of both s and d contain a path from (respectively to) the only member of the singleton seed set S_0. In directed graphs this guarantee can only be given for nodes contained in the same strongly connected component.
3. For each common landmark $l \in L$, let $p_{s \to l}$ be the path from s to l and $p_{l \to d}$ the path from l to d. Construct (by concatenation) the path $p_{s \to d} := p_{s \to l} \circ p_{l \to d}$ from s to d through l, and add it to a priority queue (sorted in ascending order by path length).
4. Return the priority queue of paths obtained in step 3.

3.1.3. Improving the Accuracy

In this section we explain our modifications to the original sketch algorithm to obtain better approximations. As a first step, we eliminate cycles in the paths found by the sketch algorithm and as a second improvement we try exploit existing shortcuts within the paths. While these two modifications to the original algorithm are simple, they provide considerable improvements in terms of the approximation quality, as we will show in the experimental evaluation.

Algorithm 4: SKETCH(s, d)

```
Input: s, d ∈ V
Result: Q – priority queue of paths from s to d, ordered by path length
1 begin
2     Load sketches Sketch(s), Sketch(d) from disk
3     L ← common landmarks of Sketch(s) and Sketch(d)
4     foreach l ∈ L do
5         p ← path from s to d through l
6         Add p to queue Q
7     return Q
```

Algorithm 5: SKETCHCE(s, d)

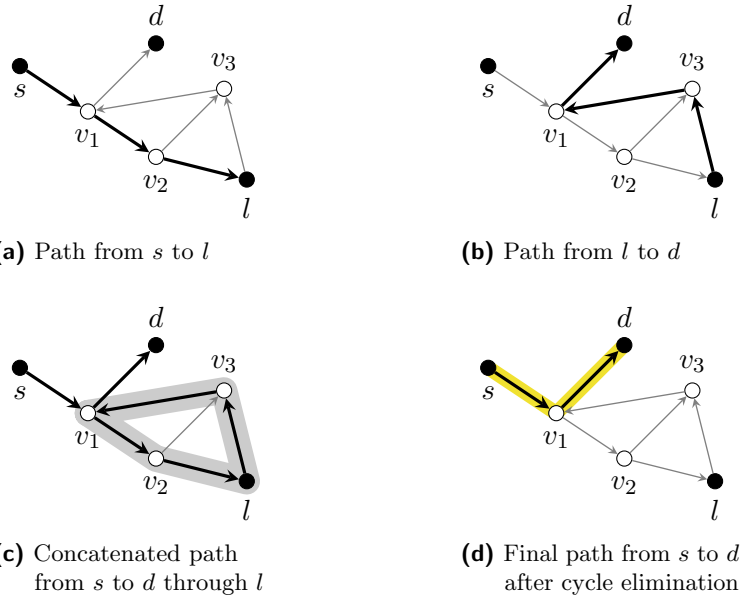
```
Input: s, d ∈ V
Result: Q – priority queue of paths from s to d, ordered by path length
1 begin
2     Q ← SKETCH(s, d)
3     foreach p = (p_1, p_2, ..., p_l) ∈ Q do
4         for i = 1 to l − 1 do
5             for j = 0 to l − i − 1 do
6                 if p_i = p_{l−j} then
7                     Q ← Q ∪ {(p_1, ..., p_i, p_{l−j+1}, ..., p_l)}
8                     break                          ▷ continue in line 3
9     return Q
```

Cycle Elimination

The paths returned by Algorithm 4 approximate the shortest path for the two query nodes and thus might be suboptimal, i. e. longer than the true shortest path. Some of the returned paths can however be easily improved, because they contain cycles. Consider the example shown in Figure 3.2: Suppose $l \in V$ is a common landmark for the nodes $s, d \in V$ and the sketches $\mathsf{Sketch}(s)$ and $\mathsf{Sketch}(d)$ contain the paths $p_{s \to l} = (s, v_1, v_2, l)$ and $p_{l \to d} = (l, v_3, v_1, d)$ respectively. Then, the queue Q returned by Algorithm 4 contains the path

$$p_{s \to l} \circ p_{l \to d} = (s, v_1, v_2, l, v_3, v_1, d).$$

Obviously, we can obtain a shorter path by removing the cycle (v_1, v_2, l, v_3, v_1), thus obtaining the path (s, v_1, d). The modified sketch algorithm, named SKETCHCE, is described in Algorithm 5.



(a) Path from s to l

(b) Path from l to d

(c) Concatenated path from s to d through l

(d) Final path from s to d after cycle elimination

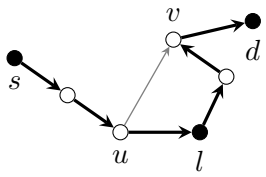
Figure 3.2.: Cycle elimination example

For a path of length l, our naïve cycle elimination routine performs at most $\mathcal{O}(l^2)$ node comparisons. In theory, we could make use of more advanced cycle elimination/detection approaches [56]. However, the diameters and thus the shortest paths in social networks are usually bounded by a small constant [72], thus we can assume constant time complexity for the cycle elimination routine on a single path. Furthermore, the number of paths in Q is bounded by the choice of precomputation rounds, k, and the number of seed sets, r. We eventually obtain the upper bound $|Q| \leq 2rk$ for the queue size because for each vertex we store two paths for every seed set (forward and backward paths). With the standard choice $r = \log n$ this leads to a time overhead of $\mathcal{O}\big(k \log n\big)$ for the cycle elimination enhancement. We can keep the queue Q in main memory, as a result the increase in running time with respect to the basic algorithm (Algorithm 4) is negligible (as we will show in the experimental evaluation).

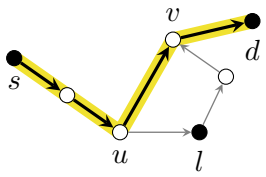
Shortcutting

The second modification we propose is path shortcutting: Suppose the queue Q returned by the algorithm contains a path $p_{s \to l \to d}$ from s to d via a landmark l. Two nodes u, v in the path might actually have a closer connection than the one contained in the respective subpath of $p_{s \to l \to d}$. Consider the example depicted below: While the nodes u and v are connected by a subpath of $p_{s \to l \to d}$ of length

3, the original graph contains the edge (u, v). We can then easily substitute this subpath by the single edge (u, v). Note that in all cases that allow for this shortcutting optimization, the landmark l will be located on the subpath from u to v.



(a) Path from s to d. The original graph contains the arc (u, v)



(b) Path from s to d after shortcutting

In order to find out whether any two nodes in a given path $p_{s\to l\to d} = p_{s\to l} \circ p_{l\to d}$ are neighbors, we start at the first vertex, s, and load the list $\mathcal{S}(s)$ of its successors in the original graph. Then we check if any of the successors of s is contained in the path $p_{l\to d}$ from the landmark l to the destination vertex d. If this is the case, we can substitute the subpath from s to this node by the single edge. If no successor of s is contained in the path from l to d, we proceed to the second node in the path $p_{s\to l}$, load the set of its successors and repeat the procedure. We can terminate this routine if we either

- find a shortcut or
- arrive at the last vertex of the path $p_{s\to l}$, the landmark node l. In this case, the original path $p_{s\to l\to d}$ cannot be improved by shortcutting, because the path $p_{l\to d}$ from l to d is already guaranteed to be a shortest path (obtained using BFS in the preprocessing step).

The complete algorithm (cycle elimination + shortcutting), called SketchCESC, is depicted in Algorithm 6.

3.1.4. TreeSketch Algorithm

In this section we describe our third contribution, a new algorithm for shortest path approximation that also utilizes the precomputed sketches.
The sketch $\mathsf{Sketch}(v)$ of a node v contains two sets of paths: (1) the set of paths connecting v to landmarks (called *forward-directed paths*) and (2) the set of paths connecting landmarks to v (called *backward-directed paths*). In the undirected setting, both sets would correspond to trees having landmarks as leaves and v as a root. Every inner node of each tree corresponds to a vertex contained in one of the paths in the sketch. In the directed setting, only the

Algorithm 6: SketchCESC(s, d)

```
Input: s, d ∈ V
Result: Q – priority queue of paths from s to d, ordered by path length
1  begin
2      Q ← SketchCE(s, d)
3      foreach p = (p_1, p_2, ..., p_{i-1}, p_i, p_{i+1}, ..., p_l) ∈ Q do
                                          ▷ p_i denotes the landmark in path p
4          for j = 1 to i − 1 do
5              S ← set of successors of p_j
6              for k = 0 to l − i + 1 do
7                  if p_{l−k} ∈ S then
8                      Q ← Q ∪ {(p_1, ..., p_j, p_{l−k}, ..., p_l)}
9                      break                          ▷ continue in line 3
10     return Q
```

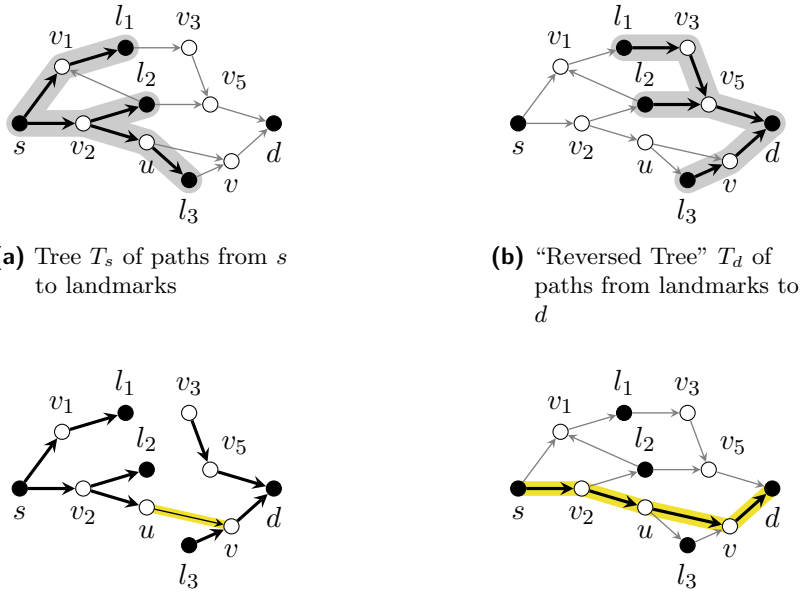
forward-directed part of the sketch (from v to landmarks) forms a tree, while the backward-directed paths yield a tree with "reversed edges" (see Figures 3.3a+b). Our new algorithm, named TreeSketch, takes the two query nodes s, d as input, loads the sketches $\mathsf{Sketch}(s), \mathsf{Sketch}(d)$ from disk and constructs the tree T_s, rooted at s, that contains all the forward paths stored in $\mathsf{Sketch}(s)$. Likewise, the "reversed tree" T_d, rooted at d, containing all the backward directed paths from the landmarks to d is being created from $\mathsf{Sketch}(d)$. Then, the algorithm starts two breadth-first-expansion on the trees simultaneously: BFS(T_s, s) from s in T_s and RBFS(T_d, d) (BFS on reversed edges) from d in T_d. At any point of time during execution, let $\mathcal{V}_{\text{BFS}}$ and $\mathcal{V}_{\text{RBFS}}$ denote the sets of visited nodes during the respective BFS runs.

For every vertex $u \in \mathcal{V}_{\text{BFS}}$ encountered during BFS(T_s, s), the algorithm loads the list $\mathcal{S}(v)$ of its successors in the original graph. Then, it checks whether any of the vertices discovered during RBFS(T_d, d) is contained in the list $\mathcal{S}(u)$. If such a vertex $v \in \mathcal{S}(u) \cap \mathcal{V}_{\text{RBFS}}$ exists (see Figure 3c), we have found a path p from s to d, given by

$$p = p_{s \to u} \circ (u, v) \circ p_{v \to d},$$

where $p_{s \to u}$ and $p_{v \to d}$ denote the paths from s to u in T_s and from v to d in T_d respectively (Figure 3d). An equivalent procedure is executed for every vertex encountered during reverse BFS from d.

We continue this procedure of BFS expansions and successor list checks, adding paths discovered along the way to the queue Q. Let l_{shortest} denote the length of the shortest path in Q. The algorithm terminates if there is no further chance



(a) Tree T_s of paths from s to landmarks

(b) "Reversed Tree" T_d of paths from landmarks to d

(c) Arc discovered in line 13 of algorithm TreeSketch

(d) Obtained path from s to d

Figure 3.3.: TreeSketch algorithm example

to find a path that is shorter than the current shortest path in Q. This is the case when the sum of depths of both BFS runs exceeds $l_{shortest}$.
The complete algorithm is depicted in Algorithm 7.

3.1.5. Implementation

We implemented all methods – Dijkstra's online shortest-path algorithms as well as the sketch-based techniques – within RDF-3X [78]. The Dijkstra's implementation is close to the one described in Chapter 2.
For our experiments, we store our social network graphs in RDF-3X edge-wise with each edge represented as a triple `(s, e, t)`. We do not restrict RDF-3X from building all the 12 indexes automatically, although we do not use all of them in this work – in fact, we exploit only SPO and OPS ordered indexes.

Sketch as an Index

We devise an additional index for storing the Sketch data structure as part of RDF-3X. Recall that the Sketch(v) of a node v consists of landmarks and paths.

Algorithm 7: TREESKETCH(s, d)

Input: $s, d \in V$
Result: Q – priority queue of paths from s to d, ordered by path length

begin
  Load $\mathsf{Sketch}(s), \mathsf{Sketch}(d)$ from disk
  $T_s \leftarrow$ tree of paths from s ▷ taken from $\mathsf{Sketch}(s)$
  $T_d \leftarrow$ tree of paths to d ▷ taken from $\mathsf{Sketch}(d)$
  $Q \leftarrow \emptyset$
  $l_{\mathsf{shortest}} \leftarrow \infty$
  $\mathcal{V}_{\text{BFS}} \leftarrow \emptyset$
  $\mathcal{V}_{\text{RBFS}} \leftarrow \emptyset$
  foreach $u \in \text{BFS}(T_s, s)$ **and** $v \in \text{RBFS}(T_d, d)$ **do**
    $\mathcal{V}_{\text{BFS}} \leftarrow \mathcal{V}_{\text{BFS}} \cup \{u\}$
    $p_{v \to d} \leftarrow$ path from v to d in T_d
    foreach $x \in \mathcal{V}_{\text{BFS}}$ **do** ▷ iteration in order of visits
      if $v \in \mathcal{S}(x)$ **then** ▷ $\mathcal{S}(x)$ is set of successors of x in G
        $p \leftarrow p_{s \to x} \circ (x, v) \circ p_{v \to d}$
        $Q \leftarrow Q \cup \{p\}$
        $l_{\mathsf{shortest}} \leftarrow \min\{l_{\mathsf{shortest}}, |p|\}$
    $\mathcal{V}_{\text{RBFS}} \leftarrow \mathcal{V}_{\text{RBFS}} \cup \{v\}$
    $p_{s \to u} \leftarrow$ path from s to u in T_s
    foreach $x \in \mathcal{V}_{\text{RBFS}}$ **do** ▷ iteration in order of visits
      if $x \in \mathcal{S}(u)$ **then** ▷ $\mathcal{S}(u)$ is set of successors of $u \in G$
        $p \leftarrow p_{s \to u} \circ (u, x) \circ p_{x \to d}$
        $Q \leftarrow Q \cup \{p\}$
        $l_{\mathsf{shortest}} \leftarrow \min\{l_{\mathsf{shortest}}, |p|\}$
    if $\text{dist}(s, u) + \text{dist}(v, d) \geq l_{\mathsf{shortest}}$ **then break**
  return Q

Therefore it may be represented as a set of triples of a form

$$\langle v \rangle \langle l \rangle \langle p_1 \dots p_m \rangle,$$

where $p_1, \dots, p_m$ constitute the path between v and the landmark l. Since G is directed, in order to take the direction of the path into account, our index structure encodes the binary *dir* value in the last bit of the landmark ID l. We treat paths as strings, map them to internal IDs, and store the sketches for all $v \in V$ in a B^+-tree index ordered by v. Then, loading of a sketch for a node basically boils down to one lookup of this node in our index followed by scan to get all the landmarks alongside the corresponding paths' IDs from disk.

Dictionary lookup of the IDs concludes the sketch loading.
Note that in our original work [40] we stored sketches directly as RDF data, thus allowing RDF-3X to create all permutations of the data on disk. Our current approach is more economical. It was shown that the disk space consumption can be further improved by avoiding dictionary mapping and storing the paths directly alongside nodes in the leaves of the B^+-tree.

3.1.6. Experimental Evaluation

In this section we provide an experimental evaluation of our algorithms. First, we give an overview of the datasets used. Afterwards, we describe the generation of test instances used in the subsequent evaluation. The experimental results include: the approximation quality of the different approaches, query running time measurements, path diversity, and the space and time requirements for sketch precomputation.

Datasets

We prove the practicability of our approach by a number of experiments on the following real-world networks:

Slashdot — a network of users of the technology-news website Slashdot, introduced in 2002. In this network, users can tag each other via friend and foe links. The data was crawled in 2008 [63].

Google Webgraph — a fraction of the webgraph released by Google for the Google Programming Contest in 2002. This network has the largest diameter of the datasets we consider [63].

YouTube — a 2007 crawl of the social network consisting of roughly 1 million users of the video-sharing community YouTube [72],

Flickr — a social network of about 1.7 million users of the photo-sharing website Flickr, crawled in 2007 [72],

WikiTalk — network of Wikipedia members commenting on each other's Talk pages. An edge exists from user a to user b if a has commented on b's user page. This network is not very well connected, only about 5% of the users belong to the largest strongly connected component [60],

Twitter — parts of the social network of the microblogging community Twitter, crawled in 2009 [82], and

Dataset	$\|V\|$	$\|E\|$	$\overline{\delta}$	$\|\mathcal{S}\|/\|V\|$	$d_{0.9}$
Slashdot	77,360	905,468	23.4	90.9 %	5.59
Google	875,713	5,105,039	11.7	49.6 %	9.02
YouTube	1,138,499	4,945,382	8.7	44.7 %	7.14
Flickr	1,715,255	22,613,981	26.4	69.5 %	7.32
WikiTalk	2,394,385	5,021,410	4.2	4.6 %	4.98
Twitter	2,408,534	48,776,888	40.5	57.5 %	5.52
Orkut	3,072,441	223,534,301	145.5	97.5 %	5.70

Datasets with no. of vertices $|V|$, no. of edges $|E|$, average degree $\overline{\delta}$, percentage of nodes in the largest strongly connected component $\mathcal{S}$ and effective diameter $d_{0.9}$ [62].

Table 3.1.: Used Datasets

Orkut — the "pure" social network Orkut, containing more than 3 million users. This network exhibits a very high average node degree [72].

The networks and their properties are listed in Table 3.1.

Methodology

In order to evaluate the approximation quality and running times of our algorithms, we use a set of test triples of the form

$$\big(x, y, \texttt{dist}(x, y)\big), \tag{3.6}$$

consisting of a pair of nodes $x, y \in V$ and the actual distance $\texttt{dist}(x, y)$ (length of shortest path) of these nodes in the graph. We generate these triples by uniformly sampling one hundred vertices and computing shortest path trees (forward and backward direction) for each vertex, using Dijkstra's algorithm [21]. As output we obtain for every sampled vertex v one tree connecting the vertex to every other reachable node and one "reversed" tree, connecting every node for which a path to the sampled vertex exists to v.
As a result we obtain a set of triples of the structure shown in equation (3.6). Then, we group these triples into categories corresponding to the distance $\texttt{dist}(x, y)$. From every such category, we sample at most 50 triples (tests) as our test set. The actual number of tests varies for every network because both the number of groups as well as the number of contained triples might be different.

Approximation Quality

In order to assess the approximation quality of the paths generated by the different algorithms, we run for every triple $\big(s, d, \texttt{dist}(s, d)\big)$ contained in the test set a shortest path query for all 4 proposed algorithms: Sketch, SketchCE,

Dataset	Tests	Sketch	SketchCE	SketchCESC	TreeSketch
Slashdot	910	46.0 %	26.0 %	0.6 %	0.00 %
Google	3,526	50.5 %	24.9 %	9.7 %	1.00 %
YouTube	758	30.0 %	12.0 %	0.6 %	0.06 %
Flickr	934	28.0 %	11.0 %	0.3 %	0.04 %
WikiTalk	780	55.0 %	31.0 %	0.2 %	0.00 %
Twitter	897	51.0 %	38.0 %	0.8 %	0.03 %
Orkut	385	71.0 %	48.0 %	0.6 %	0.10 %

Table 3.2.: Approximation quality of the shortest path algorithms

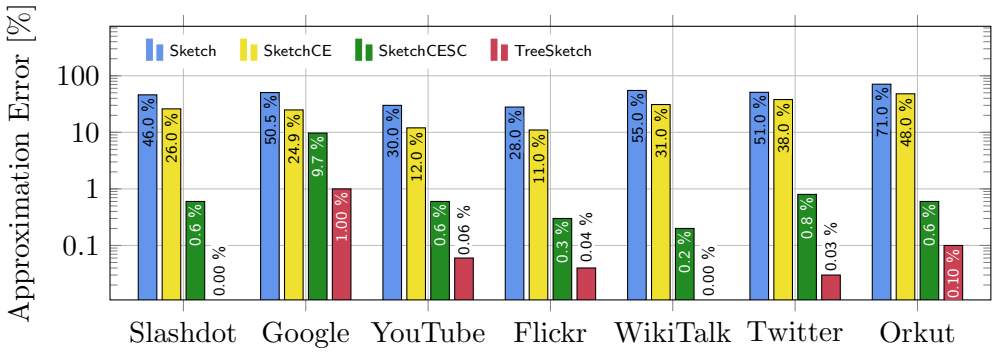


Figure 3.4.: Average shortest path approximation error `error`$(\cdot)$

SketchCESC, and TreeSketch. For every shortest path query (s, d), we compare the length l_{shortest} of the shortest path $p_{s\to d}$ in the returned queue with the true node distance $\mathtt{dist}(s, d)$ specified in the test triple. Then, we obtain the approximation error, $\mathtt{error}(p_{s\to d})$, for the path:

$$\mathtt{error}(p_{s\to d}) = \frac{l_{\mathsf{shortest}} - \mathtt{dist}(s, d)}{\mathtt{dist}(s, d)} \in [0, \infty].$$

For every algorithm we record the average approximation error over all the test triples. The obtained error values are provided in Table 3.2 and plotted in Figure 3.4, using a logarithmic scale to display the error values.
As these results clearly demonstrate, the approximation quality of the algorithms we propose in this chapter turns out to be superior to the previously proposed method (denoted as Sketch). For all datasets under consideration, we are able to return a shortest path for the query nodes with an average estimation error of 1% in the worst case, while providing exact solutions in almost all cases for the Slashdot and WikiTalk networks (using the TreeSketch algorithm). Compared to the paths returned by the basic Sketch algorithm of [22], for several datasets we are able to provide two orders of magnitude improvement in approximation quality using SketchCESC and TreeSketch. The simple cycle elimination enhancement also leads to a considerable decrease of estimation errors to the order of close to 1.7-2 factors for the datasets under consideration.

Query Execution Time

The second important evaluation category we are assessing is query execution time. We compare the results of our methods, averaged over the test triples, to the average running time of Dijkstra's algorithm. The results are listed in Table 3.3, a semilogarithmic plot of the cold cache execution times is depicted in Figure 3.5.
Observe that, using algorithm SketchCESC, we are able to answer shortest path queries with excellent accuracy within on average 190 milliseconds for the smallest dataset (Slashdot) to 4 seconds in the large Twitter dataset (cold cache). Warm cache results are all below 50 ms. Using algorithm TreeSketch we can provide even better path estimations at an negligible additional time overhead, providing the results between one and two orders of magnitude faster than the classical Dijkstra's algorithm (cold cache). Note that for the Slashdot and Wikitalk datasets, the query is executed extremely fast while achieving an approximation error of 0% for almost all of the test triples.
All query execution measurements have been carried out on an out-of-the-box laptop with a 2.0 GHz Intel Core 2 Duo processor and 4 GB of RAM, running

	Cold Cache					Warm Cache		
Dataset	Sketch	SketchCE	SketchCESC	TreeSketch	Dijkstra (Queue)	Sketch	SketchCESC	TreeSketch
Slashdot	140 ms	140 ms	198 ms	193 ms	4 s (46K)	1 ms	4 ms	5 ms
Google	932 ms	932 ms	1,270 ms	1,339 ms	35 s (157K)	3 ms	9 ms	11 ms
Youtube	872 ms	872 ms	1,282 ms	1,318 ms	48 s (380K)	5 ms	15 ms	16 ms
Flickr	1,217 ms	1,217 ms	2,177 ms	1,951 ms	73 s (696K)	5 ms	17 ms	17 ms
WikiTalk	703 ms	703 ms	1,400 ms	1,680 ms	101 s (2M)	2 ms	12 ms	13 ms
Twitter	1,932 ms	1,932 ms	3,900 ms	4,000 ms	119 s (1.1M)	5 ms	45 ms	49 ms
Orkut	1,090 ms	1,090 ms	2,595 ms	2,751 ms	503 s (2.5M)	5 ms	32 ms	37 ms

Table 3.3.: Runtime of the shortest path algorithms

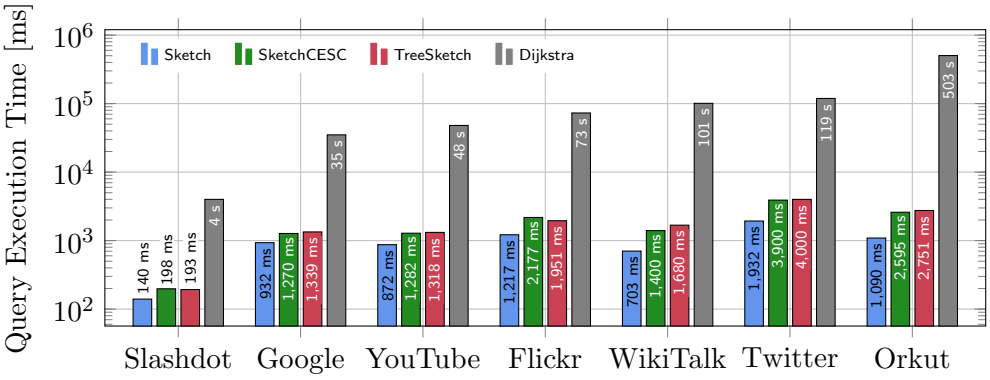


Figure 3.5.: Shortest path query execution times (cold cache)

Ubuntu Linux 10.04. For the cold cache experiments we have dropped the cache between each query.

Path Diversity

In many application settings it is not only important to quickly generate an accurate approximation of the shortest path, but also crucial to return as many candidate paths as possible. Our algorithms are designed in such a way that this goal can be satisfied.
They return an ordered queue of paths which can – for example – be used to filter out candidates based on some user-specified constraints. The average number of generated shortest paths is given in Table 3.4.

	Slashdot	Google	YouTube	Flickr	WikiTalk	Twitter	Orkut
SketchCESC	15.0	2.4	19.3	33.3	18.6	45.5	9.5
TreeSketch	31.5	3.1	40.8	55.6	50.7	92.0	29.6

Table 3.4.: Number of paths obtained

Dataset	Database	Sketch	$t_{precomp}$ [s]
Slashdot	26 Mb	36 Mb	245
YouTube	0.19 Gb	0.31 Gb	2960
Flickr	0.67 Gb	0.81 Gb	2671
Orkut	5.70 Gb	6.82 Gb	29031
Twitter	1.30 Gb	2.93 Gb	10733

Table 3.5.: Sketch Space and Time Consumption

The number of candidate paths created by TreeSketch is always greater than the number of paths generated by the other variants. For the Twitter dataset, we able to generate 92 paths on average, more than twice as much as provided by our SketchCESC algorithm.

Preprocessing

Finally, we evaluate the space and time requirements for the preprocessing step (the sketch computation).

Space Requirements. We evaluate the space consumption of the sketches by comparing their size against the size of the orginal database. See Table 3.5 for a detailed overview over the necessary disk space for the different datasets. The path sketches surpass the original database size by a factor of at most 4.

Precomputation Time. We measure the running time of the preprocessing stage for all datasets. Table 3.5 provides an overview over the required times for $k = 2$ preprocessing iterations. Note that the required time increases linearly in k. For small datasets like Slashdot, the sketches can be obtained within five minutes, while the largest dataset (Orkut) requires about 8 hours of preprocessing.
The time measurements were conducted on Dell PowerEdge M610 servers, each of which has two Intel Xeon E5530 CPUs, 48 GB of main memory, a large iSCSI-attached disk array, and runs Debian GNU/Linux with SMP Kernel 2.6.29.3.1 as an operating system.

3.2. Steiner Tree Estimation

The *Steiner tree problem*, that is, the problem of connecting a given set of nodes (also called keywords, or terminals) in a graph such that the total length is minimized with respect to some predefined cost function, is a problem with long academic history. Its importance is based on a variety of applications ranging from VLSI design to the study of phylogenetic trees. In this chapter we concentrate on three application scenarios in the area of knowledge management:

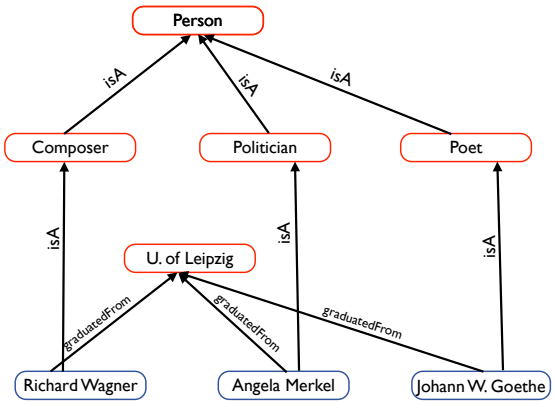


Figure 3.6.: Example of an Entity-Relationship graph

Keyword search in graphs. Keyword search is the most popular information discovery method because it does not require the knowledge of the query language or the underlying schema. Consider the entity-relationship graph where nodes are entities (e.g., extracted from Wikipedia) and edges represent relationships between entities. The keyword search problem in this setting can be formulated as: Given a few input entries, discover the relationships between them. Due to the proliferation of large web-scale knowledge bases like YAGO [96], DBpedia [74] or Freebase, the keyword search in graphs receives great attention for information discovery beyond traditional relational databases. As a concrete example, consider the query asking for the relationships between `Angela_Merkel`, `Richard_Wagner` and `Johann_W_Goethe` (see Figure 3.6). A bigger (and less informative) subtree of the graph would describe them all as belonging to the class `Person`. However, if we look for the minimal subtree connecting these three entities (the Steiner tree), we discover that all of them graduated from `University_of_Leipzig`.

Keyword search in relational databases. Since tuples can be treated as nodes connected with foreign-key relationships, the keyword search on this type of data can be again modelled as finding the Steiner trees in graphs.

Social networks. In the Social Network setting, it is important to identify familiar strangers (Stanley Milgram, 1972), i.e. the individuals who do not know each other, but share some attributes or properties in common. Here, the Steiner tree that spans the individual and its familiar strangers consists of the minimal number of edges that one needs to traverse in order to discover all of the familiar strangers.

3.2.1. Problem Difficulty

The Steiner tree problem is one of Karps 21 $\mathcal{NP}$-complete problems [53]. Moreover, it is in the $\mathcal{APX}$ class, i.e. an arbitrarily good approximation can not be achieved in polynomial time: Chebíkov and Chebíkova [19] show that no $(\frac{96}{95} - \epsilon)$-approximation can exist for any positive ϵ unless $\mathcal{P} = \mathcal{NP}$. We have to, therefore, consider approximation heuristics with rather poor theoretical guarantees.
Most of the practical approximation algorithms for this problem estimate the Steiner tree using the shortest paths between the input keyword nodes. Doing this, existing approaches usually follow one of these extreme lines:

- *No index:* Use the graph as is, and do not perform any precomputation on it. In this case, one has to follow a Breadth-First Search (or similar, e.g. Bidirectional search) expansion strategy starting from every input node of the query. While many effective heuristics on guiding and speeding up the Breadth-First Search in this setting have been proposed (BANKS [15], Bidirectional [51], STAR [54]), their performance is still poor on very large instances of graph data.

- *Index only:* Perform extensive indexing of the graph, and after that do not use the original graph at all. This, for example, includes precomputation of all keyword-to-node distances in the original graph (or, within every partition of the graph [43]). Another proposed index is based on the computing of high powers of the adjacency matrix [64]. As we see, these techniques can not easily be applied for graphs with millions of nodes

Our shortest path estimation techniques described in Section 3.1 can be viewed as a compromise between these two approaches. There, the algorithms use the precomputed index to come up with the rough approximation of the distance, and later refine it using the local search on the original graph.
Contribution. In this section we propose a mixed approach of *no index* and *only index* that allows for efficient approximation of Steiner trees for both in-memory and disk-resident graphs. The idea is to use the sketch index for the fast approximation of distances between the keywords, and then to run the (very limited) local search on both the index and the graph in order to minimize the approximation error. In other words, the goal is to generalize the approximation scheme of Section 3.1 from the case of just 2 keyword nodes (*shortest path problem*) to the general case of k nodes (*Steiner tree problem*). To show efficiency of our approach, we perform an extensive evaluation study on large real-world graphs, both in-memory and disk-resident, including YAGO (35 mln nodes) and Twitter (40 mln nodes).

Organization. After the related work overview, we describe the immediate and naive way to use the sketch index for Steiner tree approximation. Subsequently, in Section 3.2.3 we describe the new algorithm combining the sketch index with the online search on the graph. Section 3.2.4 contains extensive experiments on the real-world graphs and comparisons with other approaches.

3.2.2. Related Work

The problem has a very rich history, here we will briefly review the work done in three major directions: (i) exact methods and approximation bounds, (ii) no-index (exploration) heuristics, (iii) index-based heuristics.

Exact methods and theoretical approximations: Dreyfus and Wagner [25] and recently Ding et al. [24] exploit the dynamic programming approach to the Steiner tree problem by computing optimal results for all subsets of terminals. Both methods are naturally applicable only to moderate size graphs.
One of the first approximation algorithms is the minimum-spanning tree (MST) heuristic [58]. This heuristic builds a complete graph (a *distance network*) on terminals, where edges are attributed with the shortest distances between corresponding terminals. At the second step, the minimum spanning tree of the distance network is computed and returned as an approximation of the Steiner tree. This heuristic is guaranteed to yield the 2-approximation of the Steiner tree; this factor remained the best known until the 1990s when Zelikovsky obtained a 11/6 factor, and later the 1.55-approximation algorithm [86].

Exploration heuristics: The MST heuristic has been emulated by BANKS [15], Bidirectional [51] and [33]. BANKS operates with k iterators (one per input keyword) which are expanded in a breadth-first manner along incoming edges (i.e., in backward direction) until they meet, and then the result subtree is constructed. Bidirectional [51] improves on this method by adding the forward-directed traversal, reducing the number of iterators, and prioritizing nodes with low degrees for expansion. Both methods have an $\mathcal{O}(k)$ approximation ratio, where k is the number of input keywords.
STAR [54] follows the intuition of heuristic local search. Initially, a candidate tree is constructed by similar breadth-first expansions. Then, this candidate tree is improved by replacing the longest path in the tree with a shorter one. The algorithm terminates when no further replacement is possible. This greedy replacement strategy was demonstrated to achieve an $\mathcal{O}(\log k)$ approximation factor while significantly outperforming other exploratory heuristics.
The performance of all the exploration-based heuristics deteriorates significantly in the disk-based scenario, since the edge traversal of a disk-resident graph results in random I/O operations.

Index-based heuristics: BLINKS [43] operates on two indexes: First, for every keyword the list of nodes that can reach it is stored in the keyword-node index. Second, a node-keyword index contains the set of keywords reachable from each node, along with the corresponding distances. To avoid the expensive computation of the indexes for the large graph, the input graph is partitioned into blocks, and the top level block index as well as the intra-block indexes for each block are maintained. The keyword search then proceeds with backward expansion within multiple blocks. If the boundary of a block is reached, new iterators are created to explore the adjacent blocks. BLINKS greatly depends on the partitioning of the graph, which is quite an expensive operation in itself. Moreover, the performance of BLINKS suffers in case of relatively dense graphs (such as social networks), where two adjacent blocks may have many boundary nodes in common, and one boundary node may be shared by many blocks. This happens because BLINKS needs to create separate iterators for every such node and the block.
EASE [64] creates an index of r-radius graphs for every possible keyword of the graph. This is done by computing the r powers of the adjacency matrix of the original graph. In the online part, the precomputed r-radius graphs are retrieved and merged together, and certain nodes are pruned from the result to minimize its size. Due to inherent complexity of large matrix multiplication, we envision certain difficulties in scaling this approach for large graphs.
As we see, these methods need to have both the graph and the index in main memory and are not applicable for large, disk-resident graphs, while our techniques allow both the graph and the index to be stored on disk.

We note that some of the techniques mentioned above were designed with goals broader or different than just Steiner tree computation: BANKS, BANKS II and DPBF can also deliver the solution of the Group Steiner Tree; many of the algorithms return top-k results and rank them. A number of existing papers solve closely related problems, such as finding r-cliques [52], r-radius Steiner tree problem with EASE [64], and keyword search on RDF graphs [98].
The database community has also considered keyword queries over relational databases using the schema-based approaches: DPXplorer [4], DISCOVER [46], SPARK [68] and others (e.g.,[65]) provide algorithms based on presence of the schema. While aiming to minimize the same objective function (i.e., the size of the extracted subtree), none of these approaches can be applied directly in our schema-less graph database setting. A survey of this line of work can be found in [105].

3.2.3. Algorithms for Steiner tree estimation

We present two heuristic algorithms for the Steiner tree computation. First one is a simple extension of the SKETCH algorithm for shortest paths to the setting with k input nodes. The second algorithm can be seen as a generalization of our TREESKETCH algorithm from Section 3.1.4 for k nodes. Throughout the discussion we assume that the input graph G is connected, unweighted and undirected. The case of directed and weighted graphs is discussed at the end of the section.

Sketch algorithm for Steiner tree problem

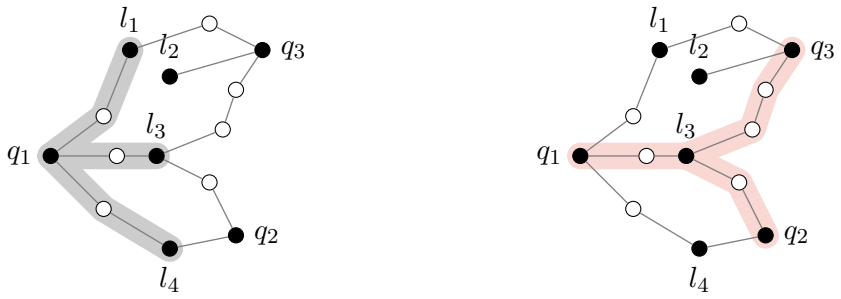
Our first contribution, the SKETCH algorithm for the Steiner tree problem, is a generalization of the sketch-based shortest path approximation to the case of k input nodes. It is depicted in Algorithm 8. There, after loading the sketches and finding common landmarks between them, we construct the Steiner tree approximation by merging the paths from the input nodes to the landmarks. Figure 3.7 illustrates this idea: l_3 is the common landmark for three sketches, so we yield the tree consisting of three paths-branches: $(q_1, \ldots, l_3), (q_2, \ldots, l_3)$ and $(q_3, \ldots, l_3)$

Statement 1. *Algorithm 8 always finds at least one approximate Steiner tree.*

This observation immediately follows from the fact that G is connected: the seed set S_0 has the single node-landmark, and this landmark is shared between all sketches, so the set L defined in line 3 of the algorithm is not empty.
Since the diameters and thus the path lengths in the considered graphs are bounded (small world phenomena in social networks [72]), we can assume the constant time of adding a path to the tree T (line 8 of Algorithm 8). Furthermore, the number of iterations (lines 4-8) does not exceed the number of different seed sets used for the sketch computation. We get $|Res| \leq k \log |V|$, and therefore the complexity of the SKETCH algorithm is $\mathcal{O}(k \log |V|)$.

SketchLS Steiner tree algorithm

In this section we describe a new algorithm, coined SKETCHLS, for the Steiner tree approximation based on sketches and local search (LS).
As outlined in Section 3.1.2, the $\mathsf{Sketch}(v)$ data structure consists of paths from v to the landmarks, so that this collection can be viewed as a tree with the root in v and landmarks as leaves. Every inner node in this tree belongs to some shortest path obtained during the precomputation stage, and the whole tree therefore is a subgraph of G.



(a) Sketches of q_1 (highlighted), q_2 and q_3 are loaded

(b) Common landmark is l_3. The result tree is highlighted

Figure 3.7: SKETCH Algorithm for Steiner tree Problem

The algorithm, depicted in Algorithm 9, starts with loading sketches and initializing k instances of Breadth-First Search that will run on sketches (lines 6-7). For every BFS process we keep the set of *frontier* nodes, that is, the nodes that are currently at the maximum distance from the source node. The BFS instances are called in a round-robin manner (lines 8-26), and the current process makes one step in the sketch (line 9). The currently visited node v may be connected in the original graph G with the nodes visited by other processes. We check whether this is the case, i.e. whether the node v is in fact a neighbor of any previously visited node (lines 14-20), by looking up the neighbors of v in the original graph G, and if this is the case, we construct the path between q_i and q_j (line 18). We keep track of the connected pairs of nodes in S_{cover}, and as soon as all the nodes are covered, stop the procedure. The set S_{cover} can be viewed as an edge list of the graph with nodes $q_1, \ldots, q_k$, since at every step we add to it an edge of a form (q_i, q_j). The condition in the lines 15-16 makes sure that this graph does not have cycles and that the result T is thus a tree.
The Figure 3.8 illustrates this algorithm. In the beginning, we load the sketches (Figure 3.8a) and initialize three BFS processes. After few iterations, the bold edges (Figure 3.8b) denote the edges already visited by Breadth-First iterators. The current node v is in the frontier of the first process, and it is connected to the node l_2 from the frontier of the third process. We immediately get the path (q_1, v, l_2, q_3). At the next step (Figure 3.8c), the remaining processes BFS_1 and BFS_2 meet via the edge (v, l_3). In this case, we also discover that v_1 and l_4 are neighbors, but that would create a cycle in S_{cover}: (q_1, q_2) and (q_2, q_1), so we skip this step and conclude: now S_{cover} covers all three input nodes.

Algorithm 8: SKETCH$(q_1, \ldots, q_k)$

Input: $Q = \{q_1, \ldots, q_k\} \in V$
Result: Res – priority queue of trees containing $q_1, \ldots, q_k$ ordered by tree size

```
1 begin
2     Load sketches Sketch(q_1), ..., Sketch(q_k)
3     L ← common landmarks of Sketch(q_1), ..., Sketch(q_k)
4     foreach l ∈ L do
5         T ← ∅
6         foreach q_i ∈ Q do
7             Add path between q_i and l to T
8         Add T to queue Res
9     return Res
```

Using the same argument as in case of the SKETCH algorithm, we see that the solution T always exists. Let us estimate the complexity of this algorithm. The size of the subgraph on which we perform our BFS processes, is the sum of k sketch sizes, i.e. $\mathcal{O}(k \log |V|)$ (As always with small-world graphs, we assume that the diameter of the graph is bounded). So, we perform $\mathcal{O}(k \log |V|)$ requests for neighbors in the graph G, and $\mathcal{O}(k^2 \log^2 |V|)$ set intersections for every pair of visited nodes in the worst case (line 14). If the maximal degree of G is D_{max}, then the hash-intersection of $F[j]$ and $\mathcal{N}(v)$ has the complexity of $\mathcal{O}(D_{max} + k \log |V|)$, and the overall complexity of SKETCHLS is $\mathcal{O}(k^2 \log^2 |V|(D_{max} + k \log |V|))$. As we see, the asymptotic behavior of SKETCHLS is worse than that of SKETCH. However, in the evaluation section we will show that it is compensated by exceptional quality of solutions found by SKETCHLS, and that both algorithms are still orders of magnitude faster than their state-of-the-art competitors. For now, we will point out the following inequality:

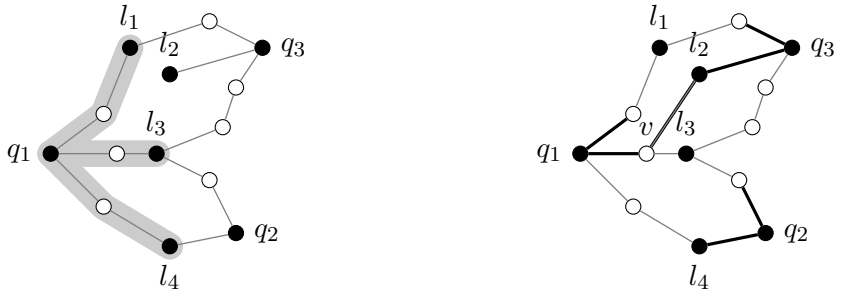
Statement 2. *Let $T_a(Sketch), T_a(SketchLS)$ and T_s be the outputs of the Sketch, SketchLS and the exact Steiner tree algorithms, respectively. Then $|T_s| \leq |T_a(SketchLS)| \leq |T_a(Sketch)|$.*

Proof. In the worst case, if the SKETCHLS algorithm does not find any shortcuts (lines 14-20, Algorithm 9), then the Breadth-First Search iterators will stop in the leaves of the sketch trees, that is in the landmarks. So, the worst-case result of SKETCHLS is equivalent to the result of Sketch: simply combine the paths between the common landmark and all the input nodes. The statement follows. □

Algorithm 9: SketchLS$(q_1, \ldots, q_k)$

Input: $Q = \{q_1, \ldots, q_k\} \in V$
Result: T –the tree containing $q_1, \ldots, q_k$

```
1  begin
2      Load sketches Sketch(q_1), ..., Sketch(q_k)
3      BFS ← new vector(k)                          ▷ vector of BFS processes
4      F ← new vector(k)                            ▷ frontiers of processes
5      S_cover ← ∅                                  ▷ set of covered nodes
6      foreach q_i ∈ Q do
7          BFS[i] ← Breadth First Search from q_i
8      foreach BFS_i ∈ BFS do
9          v ← BFS_i.next()
10         if new level of BFS_i then
11             F[i] ← ∅
12         F[i].insert(v)
13         foreach F[j] ∈ F, j ≠ i do
14             if N(v) ∩ F[j] ≠ ∅ then          ▷ N(v) = neighbors of v ∈ G
15                 if {q_i, q_j} creates a cycle in S_cover then
16                     continue
17                 n ← any node in N(v) ∩ F[j]
18                 p ← (q_i, ..., v, n, ..., q_j)          ▷ new path
19                 T.insert(p)
20                 S_cover.insert({q_i, q_j});
21         if ¬BFS_i.hasNext then
22             BFS.remove(BFS_i)
23         if BFS_i = BFS.end then
24             BFS_i ← BFS.begin
25         if S_cover covers all q_1, ..., q_k then
26             break
27     return T
```



(a) Sketches of q_1 (highlighted), q_2 and q_3 are loaded

(b) An edge between v and l_2 is discovered, q_1 is now connected to q_3

(c) BFS_1 and BFS_2 also meet (via l_3)

(d) The result tree is highlighted

Figure 3.8.: SketchLS algorithm example

Analysis of SketchLS

In this section we will give a theoretical justification of our heuristic algorithm SketchLS. Namely, we will connect it with the minimum spanning tree in the distance network [58]. We start with a few definitions.

The *distance network* D_n of the nodes $\{q_1, \ldots, q_k\}$ is a complete weighted graph where edge weights represent the distances between the nodes in our original graph G. Let us construct a subgraph T_{span} in the distance network D_n as follows: the nodes of T_{span} are $\{q_1, \ldots, q_k\}$, and the edges are all the pairs of nodes in S_{cover} at the moment when the SketchLS algorithm finishes its work. Every edge (q_i, q_j) in T_{span} is assigned with a weight w_{ij} which is equal to the distance (i.e., the number of edges) between q_i and q_j in the approximate tree T_a. The total weight of T_{span} is denoted by $w(T_{span})$, and by definition $|T_a| \leq w(T_{span})$. Note that the weight of T_{span} is computed from the result T_a of the SketchLS algorithm.

Statement 3. *The subgraph T_{span} is a spanning tree in D_n.*

Proof. It follows from the fact that T_{span} has all the nodes of D_n and contains only the edges that do not create cycles in D_n (due to the condition in Algorithm 9, line 15). Besides, it connects all the nodes in D_n (Algorithm 9, line 25). □

In addition to the spanning tree T_{span} we consider the minimum spanning tree T_{mst} in the distance network D_n. We have constructed two pairs of objects so far: a spanning tree T_{span} with the minimum spanning tree T_{mst} in the distance network D_n, and the approximate Steiner tree T_a with the exact Steiner tree T_s in the original graph G. The following statement connects them by stating that if T_{span} is a good approximation of T_{mst}, then T_a is a good approximation of T_s:

Statement 4. *If* $\frac{w(T_{span})}{w(T_{mst})} \leq 1+\epsilon$, *then* $\frac{|T_a|}{|T_s|} \leq 2(1+\epsilon)$

Proof. As demonstrated in [58], the minimum spanning tree T_{mst} is a 2-approximation of T_s, that means $\frac{w(T_{mst})}{|T_s|} \leq 2$. Now,
$\frac{|T_a|}{|T_s|} \leq \frac{w(T_{span})}{|T_s|} \leq \frac{(\epsilon+1)w(T_{mst})}{|T_s|} \leq 2(1+\epsilon)$ □

This statement does not give a theoretical upper bound for the approximation error for all types of graphs, since ϵ can be arbitrarily large. Empirically, however, the premise of this statement is easy to check, because T_{span} and T_{mst} are computed in polynomial time by SketchLS and Dijkstra's with Prim's algorithms, respectively. In other words, since the landmark-based techniques are demonstrated to perform extremely well in shortest path approximation on small-world graphs (see results in Section 3.1, but also other landmark-based approaches [22, 83, 84]), and the ϵ – i.e., the sum of the errors of k shortest path approximations – is very small (in our experiments it lies within $[0, 0.01]$ for all datasets), then our sketch-based algorithms should perform well for the Steiner tree approximation. In the evaluation section we will show that this is indeed the case.

We note that the worst-case theoretical bound for ϵ follows from the distance oracle's worst-case bound $\mathcal{O}(2c-1)$, where c is constant [22, 97]. Our ϵ is therefore upper-bounded by $\mathcal{O}(k \cdot c)$ (k times shortest path approximation error). The discrepancy between the extremely good empirical results for ϵ (and consequently, for our Steiner tree approximation heuristics) and poor theoretical bounds is an interesting topic for future theoretical research.

Weighted and directed graphs

Although our techniques are described for the case of undirected and unweighted graphs, they can be easily carried over to a more general setting.

Directed graphs. If G is a directed graph, then the precomputation procedure should distinguish between paths of two opposite directions: *from* the node

to the seed set and *to* the node from the seed set. It is achieved by running the Dijkstra's algorithm in two directions, on regular edges and on "reversed" edges. Later, the SketchLS algorithm runs the BFS on both the regular and "reversed" paths from sketches.

Weighted graphs. For the Steiner tree problem in a weighted graph G the objective function that one needs to minimize is the weight of the tree, i.e. the sum of weights of its edges. The precomputation procedure and the online heuristics remain the same, but every path now has the weight, so the sketch data structure has to contain all the weights of edges together with the sequence of nodes.

3.2.4. Experiments

In this section we present an experimental study conducted to compare our approximate Steiner tree algorithms with other approximate and exact approaches. We start with describing the competing algorithms and the datasets used for the evaluation. This is followed by the assessment of the approximation quality of different approaches and the analysis of the query size influence. Finally, we report the query runtime measurements.

Systems

We implement our algorithms for the main-memory scenario in C++ using the GNU-C++ STL library. We use the C++ implementation of DPBF [24] provided by the authors, and re-implement the Bidirectional algorithm [51] in C++ following the reference implementation in Java that was kindly provided by Heo He [43]. We also implement the minimum spanning tree heuristic (MST) and the STAR algorithm [54]. We do not consider BANKS, since its successor, the Bidirectional algorithm, outperforms it [51], and BLINKS, because it optimizes a different objective function and we are not able to measure the approximation quality.

Datasets

We examined our approach in comparison with other systems for three classes of real-world graphs:

Relational databases. We experiment with IMDb [43], a dataset derived from the popular website. It contains tuples from different tables such as *Movie*, *Person*, *Role*, connected via the foreign-key relationships.

Entity-relationship graphs. We use the YAGO dataset [96], populated with Wikipedia entities and relationships between them. With over 35 million nodes

and 40 million edges, it is the second biggest datasets that we considered.

Social networks. We consider (partial) crawls of the following networks:

Slashdot, Youtube, Flickr, Orkut — same as in Section 3.1.6

Twitter — a 2010 crawl of 40 million users of the micro-blogging network. It is the biggest graph that we considered [59].

Approximation quality

We assess our algorithms by using them for 1000 randomly generated queries per dataset. Each query has 3 to 7 keywords sampled from the graph uniformly at random. In case of smaller graphs (Slashdot, IMDb) we were able to compute an exact Steiner tree T_s using the DPBF algorithm. Then, the relative approximation error is computed as

$$\text{error}(T_a) = \frac{|T_a| - |T_s|}{|T_s|} = \texttt{error}(T_a) - 1,$$

where T_a is the size of the approximate Steiner tree returned by different heuristics in use, and $\texttt{error}(T_a)$ is the approximation ratio as defined in Section 3.1. We prefer an relative approximation error over the error ratio, since all the obtained error ratios of compared algorithms are between 1 and 2 (that is, the relative approximation error is less than 100%).

Dataset	Sketch	SketchLS	MST	Bidirect.	STAR
Slashdot	12%	4.4%	12.2%	23%	11%
IMDb	10.5%	4.1%	12%	37.9%	6.9%

Table 3.6.: Approximation error of the Steiner tree algorithms

For bigger datasets, however, the exact algorithm does not scale. The metrics of comparison is the relative difference

$$\text{diff}(T_a) = \frac{|T_a| - |T_{ls}|}{T_{ls}},$$

where T_{ls} stands for the tree yielded by the SketchLS algorithm, and T_a is the size of the tree returned by the heuristic in use. Theoretically speaking, this value can be negative. However, as in our experiments the resulting trees returned by the SketchLS algorithm are the smallest among competing algorithms, it remains non-negative for all the test cases. The obtained approximation errors are given in Table 3.6, relative difference between SketchLS and competing algorithms is provided in Figure 3.9.

Dataset	Sketch	SketchLS	MST	Bidirect.	STAR
Slashdot	7.2%	0	7.5%	17.8%	2.7%
IMDb	6.1%	0	7.6%	32.4%	5.9%
YouTube	5.6%	0	7.1%	18.7%	5.9%
Flickr	5.3%	0	6.9%	19.2%	5.1%
Orkut	7.9%	0	10.1%	25.8%	8.9%
YAGO	8.4%	0	8.3%	19.2%	8.6%
Twitter	7.4%	0	7.9%	21.5%	8.0%

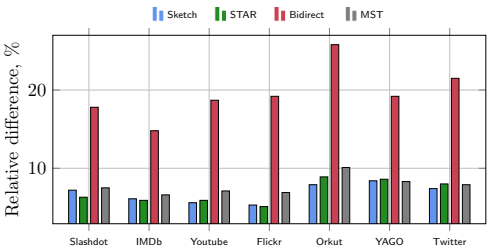


Figure 3.9.: Relative difference between SketchLS and other approaches

Query size influence

For the two datasets, Slashdot and IMDb, for which we were able to compute the exact Steiner tree, we have also measured the influence of the query size on the approximation quality. Namely, we computed the average approximation error error(T_a) separately for test queries of sizes 3..7. As Tables 3.7 and 3.8 show, the SketchLS algorithm is consistently better than any other algorithm, with the quality of approximation slightly deteriorating when the query size increases.

Query size	Sketch	SketchLS	MST	Bidirect.	STAR
$k = 3$	5.6%	2.8%	7.2%	6.8%	8.2%
$k = 4$	7.3%	3.1%	11.6%	15.9%	10.9%
$k = 5$	10.7%	4.9%	13.6%	23.8%	12.1%
$k = 6$	12.9%	5.4%	14.0%	32.9%	11.9%
$k = 7$	14.2%	5.9%	15.2%	35.5%	13.5%

Table 3.7.: Approximation error for different query sizes k, Slashdot

Query size	Sketch	SketchLS	MST	Bidirect.	STAR
$k = 3$	6.4%	3.2%	7.5%	20.5%	4.1%
$k = 4$	9.4%	4.1%	11.9%	31.7%	8.8%
$k = 5$	11.8%	5.1%	12.9%	38.5%	7.6%
$k = 6$	14.9%	6.1%	16.0%	41.2%	7.9%
$k = 7$	16.3%	6.7%	16.8%	47.7%	8.9%

Table 3.8.: Approximation error for different query sizes k, IMDb

Dataset	Sketch	SketchLS	MST	Bidirect.	STAR
Slashdot	180	359	189 K	22 K	2 K
IMDb	104	670	97 K	472 K	1862
YouTube	130	439	1.7 Mln	1.4 Mln	501 K
Flickr	133	451	1.7 Mln	1.4 Mln	659 K
Orkut	190	922	4.3 Mln	3.8 Mln	2.1 Mln
YAGO	184	1264	8 Mln	5.1 Mln	3.1 Mln
Twitter	174	1363	19.2 Mln	18.7 Mln	9.5 Mln

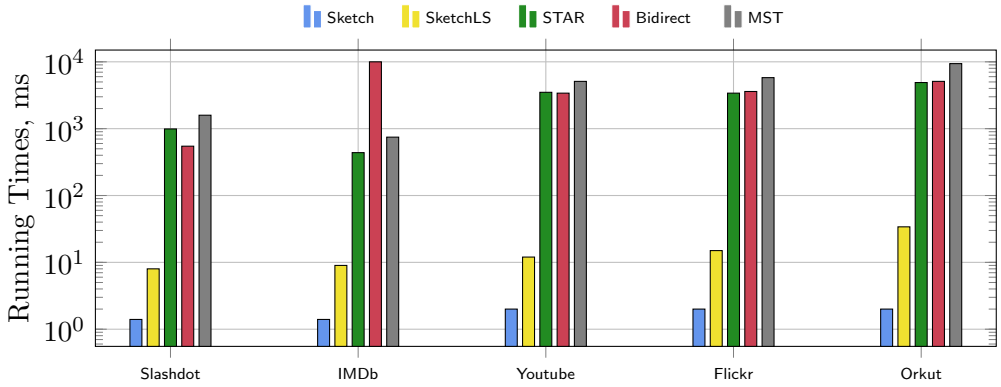
Table 3.11.: Number of visited nodes for Steiner tree algorithms

Time

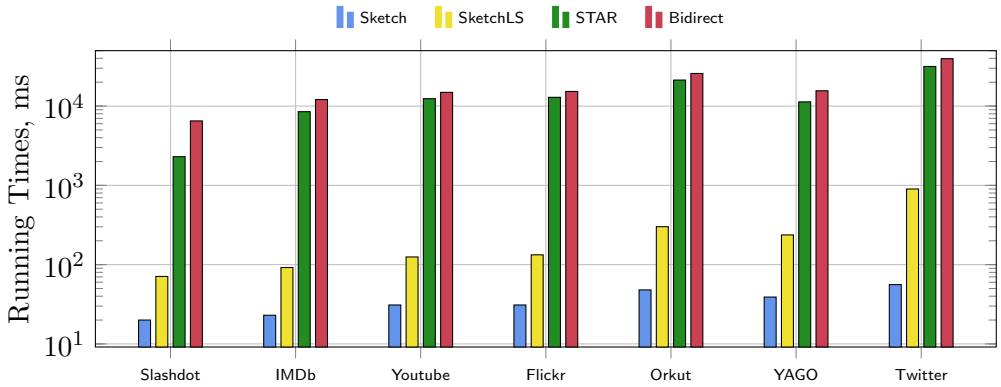
Another important factor that we measure is the number of nodes visited (touched) by the algorithms, presented in Table 3.11. Since Bidirectional and STAR are designed to return top-K results, we set that K parameter to 1 for them, so that only one result is required. We see that the Local Search done on the original graph G is limited in all cases to exploring of few hundreds of nodes. On the other hand, the *no-index* strategy of Bidirectional, STAR and MST leads to extremely large number of nodes that they visit during the query processing. This, in turn, explains the prohibitively large running times for the in-memory and especially disk-resident graphs, presented in Table 3.9 and plotted in Figure 3.10 using a logarithmic scale on the vertical axis. Note that our runtime system uses caching, that makes the difference between in-memory and on-disk performance smaller than it could be, but the effects of random disk access (e.g., when reading a list of neighbors) are still significant.
Two of our biggest datasets, YAGO and Twitter, are only used for the disk-based evaluation. The rest of the datasets is included in both in-memory and on-disk evaluation.
As the numbers demonstrate, our algorithms' running times are clearly superior to all existing approaches: the simple Sketch algorithm is up to 3 orders of magnitude faster than any other existing algorithm under consideration, the new SketchLS algorithm is up to two orders of magnitude faster, in both the main-memory and disk-based scenarios.
All the algorithms were run on a commodity server with the following specifications: Dual Intel X5570 Quad-Core-CPU, 8 Mb Cache, 64 Gb RAM, 1 Tb SAS-HD, Redhat Enterprise Linux with 2.5.37 kernel.

Table 3.9.: Running Times

	In-memory						On-disk			
Dataset	DPBF	MST	Bidirect.	STAR	Sketch	SketchLS	Bidirect.	STAR	Sketch	SketchLS
Slashdot	38.2 s	1.59 s	547 ms	990 ms	1 ms	8 ms	6.5 s	2.3 s	20 ms	71 ms
IMDb	21.5 s	747 ms	10 s	438 ms	1 ms	9 ms	12.1 s	8.5 s	23 ms	92 ms
Youtube	–	5.1 s	3.4 s	3.5 s	2 ms	12 ms	14.9 s	12.4 s	31 ms	125 ms
Flickr	–	5.8 s	3.6 s	3.4 s	2 ms	15 ms	15.3 s	12.9 s	31 ms	133 ms
Orkut	–	9.4 s	5.1 s	4.9 s	2 ms	34 ms	25.8 s	21.3 s	48 ms	301 ms
YAGO	–	–	–	–	–	–	15.6 s	11.3 s	39 ms	237 ms
Twitter	–	–	–	–	–	–	39.6 s	31.6 s	56 ms	900 ms



(a) In-memory graphs



(b) Disk-resident graphs

Table 3.10.: Running times of the Steiner tree approximations

CHAPTER 4

Optimization of Complex Graph Queries

Parts of this chapter were published in [37]

In the previous two chapters we have discussed the two specific types of graph queries, shortest path and reachability queries. Now we will consider a very general class of graph pattern matching problems expressed as SPARQL queries, and the query optimization problems related to this class of queries.

As the quality of the existing graph datasets improves, users tend to formulate interactive graph pattern matching queries of increasing complexity, just like it is commonplace with SQL in modern RDBMSs. For example, the query log of the interactive DBpedia endpoint has SPARQL queries with up to 10 joins [11], and analytical queries in the biomedical domain can include more than 200 joins [88]. Of course, queries of this size do not usually come from users directly, but are rather generated by user-facing interactive tools (e.g., large provenance queries of [88] are generated from a higher lever declarative language), and therefore their size can be arbitrarily large.

In this section we describe a novel join ordering algorithm aimed at large SPARQL queries. Essentially, it is a SPARQL-tailored query simplification procedure that decomposes the query graph into star-shaped and chain-shaped subqueries; these subqueries correspond to matching star-shaped subgraphs and chains of stars. For these subqueries we introduce a linear-time heuristical join ordering algorithm that takes into account the underlying data correlations to construct an execution plan. The simplified query graph typically has a much smaller size compared to the original query, thus allowing to run Dynamic Programming on it. In order to estimate the cardinalities in the simplified query, we introduce new statistical synopsis coined Characteristic Pairs.

We start with considering the query optimization challenges of graph pattern matching queries; most of these challenges are caused by the nature of RDF data (Section 4.1). Section 4.2 describes our novel algorithm for star-shaped SPARQL query optimization. In Section 4.3 we show how this algorithm can be incorporated into the query optimizer for queries of a general shape, as well as present a novel statistical synopsis for accurate cardinality estimations beyond

star-shaped queries. To validate the effectiveness and efficiency of our join ordering strategy, we run our join ordering algorithms on thousands of generated queries over real-world datasets, and compare them with the state-of-the-art SPARQL query optimization algorithms in Section 4.4. Finally, Section 4.5 discusses related work.

4.1. Motivation

Finding the right execution plan (basically, the right join order) is known to be a challenging task for any query optimizer. In RDF systems, additionally, the optimizers are faced with extremely large sizes of queries due to verbosity of the data format, and with the lack of schema that challenges the cardinality estimation process, an essential part of any cost-based query optimizer. It is easy to construct a query with less than 20 triple patterns whose compilation time (dominated by finding the optimal join order) in the high-performance RDF-3X system [78] is one order of magnitude higher than the actual execution time. Query 4.1 contains a SPARQL formulation of such a query against the YAGO2 knowledge base; it returns the information about a German writer, the novels he created, and further art works linked to them. On the other hand, another popular high-performance triple store, Virtuoso 7, seems to spend much less time finding the join order (probably employing some kind of greedy heuristics), but pays a high price for the (apparently) suboptimal ordering. For that specific Query 4.1 we have measured the following compile and runtimes in two systems:

System	Query Compilation (dominated by query optimization)	Query Execution
RDF-3X	78 s	2 s
Virtuoso 7	1.3 s	384 s
This work	1.2 s	2 s

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

This problem becomes even more pressing, as emerging applications require the execution of queries with 100+ triple patterns [88]. One of the popular alternatives for Dynamic Programming for such queries – Greedy heuristics - faces a hard challenge of greedily selecting even the first pair of triples to join due to structural correlations between different triple patterns [76]. Indeed, a triple pattern can be quite selective itself (e.g., *people born in France*), but not

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

Query 4.1: Complex graph pattern matching query over YAGO2. We use the standard shortcut `a` for `rdf:type`

within the considered group of triple patterns (that could describe, e.g., *French Physicists*).
In the following section we present a detailed analysis of query optimization challenges caused by large graph pattern matching SPARQL queries.

Analysis of challenges

Finding the optimal join order of a SPARQL query is very challenging due to the nature of RDF data. The origins of the technical problems can be roughly split into two parts: the size of the search space, and the lack of schema.
Search space size. The RDF triple format is very verbose. Thus, for example, the TPC-H Query 2 written in SPARQL contains joins between 26 index scans (as opposed to joins between 5 tables in the SQL formulation!). The number of possible query plans is in the order of factorial of the table number, i.e. $5! = 120$ plans in SQL vs $26! = 4 \cdot 10^{26}$ plans in SPARQL. This is a price that the engine has to pay for the flexibility of the data schema. Therefore, the sheer size of the search space for large SPARQL queries does not allow the standard Dynamic Programming exploration of equivalent query plans, since it has to look at all the valid plans in order to find the cheapest one.
Lack of schema. The lack of schema – a typical situation for any graph database – leaves the optimizer without essential information that is readily available to any relational optimizer, such as the set of tables, their attributes, primary and foreign keys. For example, a relational system would have information that an entity of a type `Person` has attributes `Name`, `Birthday`, `Birthplace`

(a) SPARQL subgraph

Partial Plan	Est.Size
$(?s, created, ?piece) \bowtie (?piece, linksTo, ?city)$	1.5 Mln
$(?piece, linksTo, ?city) \bowtie (?city, hasLatitude, ?lat)$	1.3 Mln

(b) Entries of the DP table

Figure 4.1: Part of the Dynamic Programming table for Query 4.1

etc., and foreign key relationships to other entities (`Places`, `Companies` etc). The relational optimizer can therefore keep the statistics on these attributes and foreign keys (e.g., an average person has lived in 3 different places) and use it for result size estimation. All this information is only *implicitly* present in RDF data, where attributes and foreign keys create *structural correlations* in the RDF graph. In other words, some of the predicates tend to occur together as labels of outgoing edges of the same node (e.g., `bornIn`, `hasName` and `created`), and some subgraphs tend to occur together in RDF graphs (e.g.,writers and books). The simplest of these correlations – corresponding to the attributes of the same entity – are captured by Characteristic Sets (CS) of the RDF graph [76], described in Chapter 1. However, the Dynamic Programming (DP) algorithm for join ordering requires computing the CS-based cardinality estimates for every non-empty subgraph of the query. It significantly increases the (already high) runtime of the DP algorithm, as we will demonstrate in the evaluation section. Moreover, CS do not capture correlations between different subgraphs in the RDF graph. In the absense of this information, the optimizer has to rely on the independence assumption and fails to estimate the result sizes of most of the partial plans.

One more issue with Dynamic Programming is that, even for the mid-sized query graphs, the algorithm ignores the structure of the query, and therefore considers a lot of a priori suboptimal subplans during the plan construction.

To illustrate the last point, consider a subgraph of Query 4.1, given in Figure 4.1a, and some of the corresponding subplans, generated by the DP (depicted, together with estimated result sizes, in Figure 4.1b).

Here, the independence assumption leads to a significant underestimation of the size of the result, since the optimizer merely multiplies the frequencies of two predicates. Take the first partial plan given in Figure 4.1b. Its cardinality — given the independence assumption – is computed based on multiplication of of the frequencies of `created` and `linksTo`, and results in a somewhat modest amount of 1.5 million tuples. In reality, however, these two predicates are not independent: artifacts created by people (books, movies, songs) tend to link to multiple entities mentioned in them or related to them. The real cost of the

partial plans is therefore much worse than what the optimizer expects. Indeed, the first partial plan returns all the people that have created anything that links to any entity (hundreds of millions of entities), and the second subplan yields all the entities linked to each other, such that one of them has a latitude as an attribute. Clearly, performing these chain-shaped joins earlier during query execution would produce enormous intermediate results. However, completely avoiding chain joins can not be made the "rule of a thumb" either, since some of the chains may yield extremely small results. The (nearly optimal) join ordering strategy is to split the query into star- and chain-subqueries while still keeping the correlations between different subqueries, and this is the strategy that we pursue in this chapter.

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

Note that, although we concentrate on triple stores in this chapter, the problem of join ordering is orthogonal to the underlying physical storage and is therefore common to all the RDF systems. The sources of this problem lie in verbosity of the RDF data model and high irregularity of real-world datasets. Our solutions (join ordering algorithms and statistical data structures for cardinality estimation) do not assume any particular organization of a triple store (i.e., specific indexes, data partitioning etc.) and are therefore applicable to a wide range of systems.

4.2. Star Query Optimization

We first consider finding the optimal join order for queries of a particular shape, namely the star-shaped queries. We start by introducing a statistical data structure coined the hierarchical characterisation of the RDF graph, and then

describe the algorithm that employs it to find an ordering of joins in star queries.

Hierarchical Characterisation

A characteristic set, defined in [76] as a set of outgoing edge labels for the given subject, tend to capture similarity between entities described in the RDF dataset. Thus, entities with the characteristic set {`hasTitle`, `hasAuthor`} usually describe books. While the lack of fixed schema prevents us from assigning these entities a type "Books" (e.g., we are not going to store such entities in a separate table), the characteristic sets allow us to predict selectivities for given sets of query triple patterns.

In real-world RDF datasets, set inclusion between different characteristic sets also bears some semantic information. Consider the following Characteristic Set (CS) that describes a type `Writer`: $S_1 = \{\texttt{hasName}, \texttt{bornIn}, \texttt{wroteBook}\}$, and another CS that characterizes entities of type Person: $S_2 = \{\texttt{hasName}, \texttt{bornIn}\}$. Note that $S_2 \subseteq S_1$, and at the same time `Writer` is a subtype of `Person`. We find this situation to be extremely frequent in knowledge bases. For instance, the majority of characteristic sets in YAGO [45] are subsets of each other, reflecting the class hierarchy of YAGO entities.

Along with the set of predicates, the CS keeps the number of occurrences of this predicate set in the dataset [76] (denoted as $count(CS)$). Here we introduce a generalization of this measure, namely the aggregate characteristic of CS $cost(CS)$. It is defined as the sum of occurrences of all the supersets of the given CS:

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

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

```
select distinct ?s
where {?s p1 ?o1. ... ?s pk ?ok.}
```

Second, *cost* can be applied to any set of predicates that does not form the CS. For example, the set of two predicates $S = \{\texttt{hasName}, \texttt{wroteBook}\}$ is not characteristic, since these two predicates are always accompanied by `bornIn`. However,

$cost(S)$ still can be used to estimate the size of the join of two corresponding triple patterns, namely $(?s, hasName, ?name)$ with $(?s, wroteBook, ?book)$.
The following obvious property holds for two sets S1, S2 such that $S_2 \subseteq S_1$: $count(S_2) \geq count(S_1)$. For instance, the number of entities of type `Person` is clearly not smaller than the number of `Writers`. Same holds for the costs of sets in this situation: $cost(S_2) \geq cost(S_1)$.
At the same time, some subsets of the predicate set S may be cheaper than others. Consider again $S = \{\texttt{hasName}, \texttt{bornIn}, \texttt{wroteBook}\}$ and all its two-element subsets along with their costs listed in the table below:

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

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

Definition A *Hierarchical Characterisation* of a dataset R is a collection $\{H_0, \ldots, H_k\}$ of sets H_i, such that

1. H_0 is the set of all characteristic sets of R
2. $H_i = \{ \underset{\forall C \subset S \wedge |C| = |S|-1}{\arg\max} cost(C) \mid \forall S \in H_{i-1}\}$. In other words, for every set S in H_{i-1} we find the cheapest (w.r.t. *cost*) subset of S of size $|S| - 1$, and include it into H_i.
3. every $S \in H_{i-1}$ stores a pointer to its cheapest subset $C \in H_i$
4. $\forall S \in H_k : |S| = 2$

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

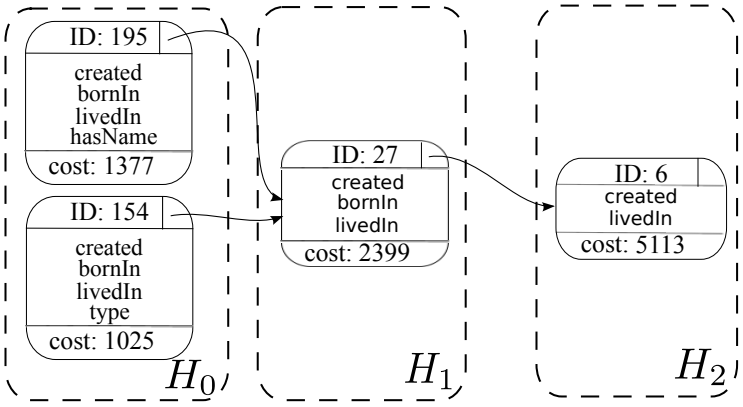


Figure 4.2.: Hierarchical Characterisation

An example of the part of the Hierarchical Characterisation for the YAGO2 dataset [45] is given in Figure 4.2. There, two characteristic sets in H_0 (IDs 195 and 154) point to the same cheapest subset from H_1.
As we have noted, in real RDF datasets most of characteristic sets are subsets of each other. This leads to the same sets appearing in different levels of the HC: both as leaves in the H_0 level, and as subsets of other sets in some H_i level. Naturally, we do not need to store the same set twice, so this ambiguity stays only on conceptual level. In theory, if H_0 level has m different sets drawn from n distinct elements, we can get up to $m \cdot n$ distinct subsets in HC. In practice, however, due to the fact that the same sets are shared between different levels, and different sets can get the same cheapest subsets, this number is quite close to m.

Computing Hierarchical Characterisation

A straightforward computation of Hierarchical Characterisation could be organized as follows: starting with Characteristic Sets, at each iteration for every set we find all its subsets (that are one element smaller), get the cheapest one, pass all the cheapest subsets for each set to the next iteration. The iterations should repeat until all the newly generated sets are of size 2. This however, generates the same set many times, since (a) the sets may appear in multiple levels of HC, (b) two different sets may have the same cheapest subset, as it is the case with sets 195 and 154 in Figure 4.2.
The computation that avoids this pitfall is sketched in Algorithm 10. It works as follows. First, the Characteristic Sets are computed [76]. They are ordered by increasing size, starting with one element sets. Then, the iterations computing subsets run until no new subsets appears (lines 3-5 in Algorithm 10). At every

iteration, we run two iterators S_1 and S_2 over the sets under consideration. For every pair of S_1, S_2 we check if $S_2 \subset S_1$ and $|S_2| = |S_1| - 1$, and keep the cheapest S_{best} of all such subsets (lines 10-16). Additionally, we need to consider the subsets of S_1 that are not in our list of sets, i.e. the subsets that are not characteristic sets. In order to enumerate such subsets, we employ the Bankers enumeration of sets [67]. The Bankers enumeration iterates over subsets of the set of all predicates in graph R, and emits the ones that are not characteristic sets. The reason we choose the Banker's enumeration is that it lists all the subsets in monotonically decreasing size, so our enumeration can stop earlier (we only need the subsets that are one element smaller than the set S_1).

Algorithm 10: Computing Hierarchical Characterisation

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

```
1  begin
2      Sets ← computeCharSets()                    ▷ see algorithm in [76]
3      H ← ∅
4      while Sets ≠ ∅ do
5          H ← H ∪ Sets
6          Sets ← computeSubsets(H)
7      return H

8      PRECOMPUTATION
9      SubsetIterator si ← init Banker's iterator
10     res ← ∅
11     foreach S1 ∈ Sets do                        ▷ iterate in increasing set size
12         S_best ← ∅
13         foreach S2 ∈ Sets : |S2| ⊂ |S1| & |S2| = |S1| − 1 do
14             if cost(S2) < cost(S_best) then
15                 S_best ← S2
16             while si ≠ S2 do                    ▷ get all subsets of S1 that are not in Sets
17                 Si ← si
18                 if |Si| = |S1| − 1 & cost(Si) < cost(S_best) then
19                     S_best ← Si
20                 si ← si.next
21     store the pointer to S_best in S1
22     if S_best ∉ Sets then
23         res.insert(S_best)
24     return res
```

Consider the following fragment of computation, where sets are listed in order of increasing size; sets printed in bold are characteristic, the other sets are the "missing" sets generated by the Bankers iterator si:

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

Suppose that the iterator si is enumerating sets from $\{1, 2, 3\}$ to $\{1, 3, 5\}$, until it reaches S_2 (line 15 of Algorithm 10). The last encountered set is $\{1, 3, 5\}$, which is not characteristic (i.e., does not belong to the input $Sets$), but is a subset of S_1. If it is cheaper than the previously considered S_2, we update the S_{best} variable (line 16).
Although the Bankers iteration potentially enumerates all the subsets of all predicates in the dataset, in reality it stops relatively early, since it is always bounded by the largest set in $Sets$ (see condition in line 13). This largest set gets smaller with every iteration, since every iteration considers subsets of sets generated in the previous iteration. Additionally, the biggest portion of the HC is identified during the first iteration, and the process converges really quickly. We also note that the set inclusion check in the innermost **for** loop (lines 13-20) is implemented extremely efficiently using Bloom filters. In our experiments, computing the Hierarchical Characterisation of the UniProt dataset (with over 850 million tripes stored on disk) is done within 6 iterations and takes ca.700 ms.

Join Ordering for Star Queries

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

```
select *
where {?s p1 ?o1. ... ?s pk ?ok.}
```

Let $S = \{p_1, \ldots, p_k\}$ be the corresponding set of predicates. Our main idea is to extract the part of the Hierarchical Characterisation of the dataset starting with the set S. Namely, we find the set $S_1 \in H_0$ such that $S_1 = S$, get its cheapest subset S_2 (remember that S_1 has a pointer to S_2) and find out the predicate p in

$S_1 \setminus S_2$. This predicate `p` corresponds to one of the triple patterns of the query `(?s, p, ?o)`; we put this triple pattern to be the last in the join order. The procedure repeats with S_2: follow the pointer to its cheapest subset S_3, put the triple pattern with the predicate from $S_2 \setminus S_3$ to be the last but one predicate in the join order. The process terminates when the current set S_k has only two elements: these predicates correspond to triple patterns that will be joined first.

Algorithm 11: Join Ordering for Star Queries

Input: SPARQL star-shaped graph Q^R
Result: O – Join Ordering for Q^R

```
1  begin
2      P ← {p_1, ..., p_k}                          ▷ set of predicates in Q^R
3      S ← getHierarchicalChar(P)
4      O ← [ ]
5      while S.size > 2 do
6          S_sub ← S.subset                          ▷ pointer to the cheapest subset
           ▷ must hold: |S_sub| = |S| − 1, by definition of HC
7          p ← S \ S_sub                             ▷ most "expensive" predicate in S
8          O.append(p)                               ▷ append to the front of the list
9          S ← S_sub
       ▷ the two elements left in S = {p_1, p_2}
10     if Q^R has constants : ∃(s_i, p_i, c_i) ∈ Q^R then
11         if one of the constants is a "key" then   ▷ see definition of a "key" in text
12             swap the corresponding predicate p_i in O with the O.front
13         else
14             keep pushing p_i to the front of O, while
15             cost(IndexScan) < cost(⋈)
16     return O
```

The pseudo-code of the algorithm is depicted in lines 1-9 of Algorithm 11. We note that if there is no set in H_0 that contains all the predicates from P (i.e., the lookup in line 3 fails), then the corresponding query yields an empty result. The intuition behind this approach is the following: starting from the set of triple patterns, we find out what is the most expensive triple pattern, and schedule it to be the last in the join order. This expensive triple pattern is exactly the one that does not appear in the cheapest subset (w.r.t. our cost function) of the predicate set. Following this logic at each step, we construct the join tree top-down.

An illustration of this algorithm is given in Figure 4.3. The set with ID 154 (see Figure 4.2) has the predicates from all four given triple patterns. Its cheapest subset in the HC is the set with ID 27. Therefore, the triple pattern with the

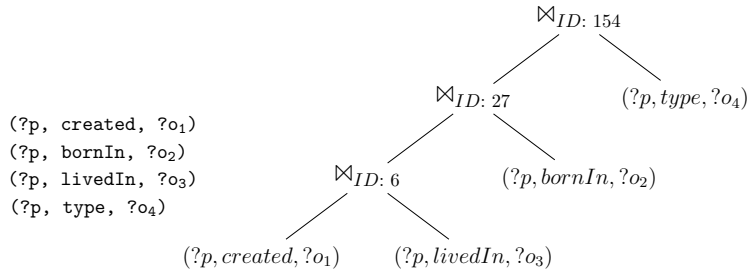


Figure 4.3.: Triple patterns and their optimal ordering (joins are annotated with IDs of corresponding Char.Sets from Figure 4.2, page 92)

predicate `type` will be the last one in the join order (the upper-most triple pattern in the join tree). Similarly, the cheapest subset of the set with ID 27 is the set with ID 6, and the missing predicate in it is `bornIn`. Since the last set has two predicate, they form the first join in the join ordering (lower-most level in the tree).

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

Unfortunately, this no longer holds if the query has constant objects, i.e. when some of $?\mathtt{o}_1, \ldots, ?\mathtt{o}_k$ are replaced with literals or URIs. We have to, therefore, rely on a heuristics. It seems impossible to precompute all the correlations between constant objects and predicates in all sets of Hierarchical Characterisation. However, in real datasets we observed that some predicates in the sets of HC are extremely selective (like keys in relational world), and then all other predicates nearly functionally follow the selective ones. In our example with books and people, the name of the author is nearly the key (same with the title of a book). This can be captured while constructing the HC, if we track the multiplicity of each predicate in the sets. Then, the key predicates are those with the multiplicity of 1. Note that we only need to store this multiplicity information for the sets from H_0, i.e. the characteristic sets.

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

1. one of the bounded objects is in the triple with the key predicate (lines 11-13). The entire star query is therefore a lookup of properties of a specific entity. We push down this triple pattern (basically by appending it at the front of our join ordering), and stop.

2. otherwise, we keep pushing down the constants in the join tree and stop when the cost of the corresponding index scan is bigger than the cost of the join on that level of the tree (lines 14-15)

We do not want to simply push down all constants, since some of the object constants (especially for the type predicate) are quite unselective and it is possible that the lowest join in the tree produces less tuples than the index scan on such an unselective constant alone.
So far we have considered star queries centered around subject. Such patterns are extremely common in RDF datasets and are prevalent among the queries. However, the same techniques work for stars around objects (i.e., based on the object-object join). Since these queries are still valid SPARQL, the system derives and stores Hierarchical Characterisations for object-centered stars, too.

4.3. Query Optimization of General Queries

In this section we describe the algorithm for join ordering in general queries expressed in SPARQL 1.0. Our main idea is to decompose the query into star-shaped subqueries connected by chains, and to collapse the subqueries into meta-nodes. The star-shaped subqueries are optimized by Algorithm 11 from Section 4.2. Then, the Dynamic Programming algorithm is run on top of the simplified query. In order to enable accurate cardinality estimations in the simplified query, we introduce a novel synopsis (Characteristic Pairs) that captures structural correlations in the RDF graph beyond star subqueries.

Characteristic Pairs

While some of the correlations between triples are captured by Characteristic Sets (define types in the RDF dataset) and consequently Hierarchical Characterisation (inheritance between different types, subject to a specific cost function), we are still missing other relationships between different types.
Let us illustrate it with an example. Consider the triples describing the person and its birthplace:

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

There, the object id s_2 in the triples describing the person is used to link it to the city. In a way, this correspond to the "foreign key" concept in relational databases, except that of course RDF does not require to declare any schema. Mining these foreign keys thus becomes a challenge for the system. Knowledge of such dependencies is, on the other hand, extremely useful for the query

optimizer: without it, the optimizer has to assume independence between two entities linked via `bornIn` predicate, thus almost inevitably underestimating the selectivity of the join of corresponding triple patterns. In reality, almost every person has information about the place of birth, so the selectivity of the join is close to 1.
In order to capture these "foreign key"-like relationships between nodes in the RDF graph, we will store pairs of characteristic sets that occur together (i.e., are connected by an edge) in the RDF graph, along with the number of occurrences.

Definition Let $S_c(s)$ denote a characteristic set of edges emitting from a subject s (in other words, $S_c(s)$ is a type of s). We define a *Characteristic Pair* between two characteristic sets for s and o in the dataset R as

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

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

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

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

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

(where the denominator captures the number of occurrences of Characteristic Set for s), tells us, to how many objects of the same type does the subject s connect.
Although in theory, with n distinct characteristic sets we can get up to n^2 characteristic pairs, in real datasets only few pairs appear frequently enough to be stored. For the YAGO-Facts dataset (the core of YAGO2), out of ca. 250000 existing pairs, only 5292 pairs appear more than 100 times in the dataset. This way, the frequent characteristic pairs for YAGO-Facts consume less than 16 KB. We do not store the pairs with count being smaller than 100. For such pairs, the independence assumption provides a "close enough" estimate of the result size. The optimizer will, most likely, underestimate the size to be just 1 tuple, but we found that misestimation in such cases does not lead to a plan whose performance would differ from the optimal one.

Estimating Cardinalities using Characteristic Pairs

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

```
select distinct ?s ?o
where {  ?s p1 ?x1 .
         ?s p2 ?x2 .
         ?s p3 ?x3 .
         ?o p4 ?y1 .
   }
```

In order to estimate the result size of the query, we simply need to find all the characteristic pairs consisting of sets that contain the predicates $\texttt{p}_1$, $\texttt{p}_2$, $\texttt{p}_3$ and $\texttt{p}_4$:

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

where C_1 and C_2 are the characteristic sets that contain $\texttt{p}_1$, $\texttt{p}_2$, $\texttt{p}_3$ and $\texttt{p}_4$, respectively.

Note that this computation does not assume independence between any predicates, and works for stars of arbitrary size. However, this simple estimation is only possible due to the `distinct` keyword: in general, the query can produce duplicate results for ?s and ?o for two reasons: first, there may be multiple bindings for $?\texttt{x}_1$, $?\texttt{x}_2$ and $?\texttt{y}_1$; second, there may exist multiple bindings of ?o for the same ?s.

In order to cope with the first issue (multiple bindings for objects), predicates in characteristic sets are annotated with their number of occurrences in entities belonging to this set [76]. Similarly, we annotate the predicate that connects two entities in the characteristic pairs with its number of occurrences. Formally, for $P = (S_c(s), S_c(o), p)$ we compute $count(p) = |\{(s,p,o) \mid (s,p,o) \in R \wedge P_C(S_c(s), S_c(o)) = P\}|$. In other words, based on $count(p)$ we can derive whether the "foreign key" relationship between s and o is a "one-to-one" or "one-to-many". Namely, if $count(p) = count(P_C(S_c(s), S_c(o)))$, it is one-to-one, and with $count(p) > count(P_C(S_c(s), S_c(o)))$ it is one-to-many.

Consider again the query above but without the `distinct` modifier, and suppose that characteristic sets for ?s and ?o, and the characteristic pair are as depicted in Figure 4.4. The first column of each table gives us the number of distinct stars and pairs of stars. The rest of the columns contain the *count*s (number of occurrences) of predicates in characteristic sets and characteristic pairs. This means that, on average, one entity of type $S_c(s)$ has $\frac{1000}{1000} = 1$ predicate $\texttt{p}_1$, $\frac{3000}{1000} = 3$ predicates $\texttt{p}_2$ and $\frac{2000}{1000} = 2$ predicates $\texttt{p}_3$, and on average for every s

distinct	p_1	p_2	p_3
1000	1000	3000	2000

(a) CS for ?s

distinct	p_4
5000	5100

(b) CS for ?o

distinct	p_3
1000	2000

(c) P_C(?s, ?o)

Figure 4.4.: Representation of two Characteristic Sets and one Characteristic Pair

there are $\frac{2000}{1000} = 2$ occurrences of o connected to it via p3. We can therefore estimate the cardinality of the query without the `distinct` keyword as:

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

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

Join Ordering for General SPARQL Queries

Our join ordering strategy for general SPARQL queries is given in Algorithm 12. The algorithm starts with clustering the query into disjoint star-shaped subqueries (lines 12-24). In order to do it, we order the triple patterns in the query by subject (line 14), and group triple patterns with identical subjects (line 15). These groups potentially form star-shaped subqueries. Then, for every group of triple patterns we estimate its cardinality using characteristic sets. If its small enough (in our experiments we used a cutoff of 100K tuples), the group is turned into star subquery and the corresponding edges are removed from the query graph (lines 18-20).
Consider the query from Figure 4.5a as an illustration. Its triple patterns after grouping look as follows:

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

The corresponding subgraphs of the SPARQL query graph are denoted as P_1

Algorithm 12: General SPARQL join ordering algorithm

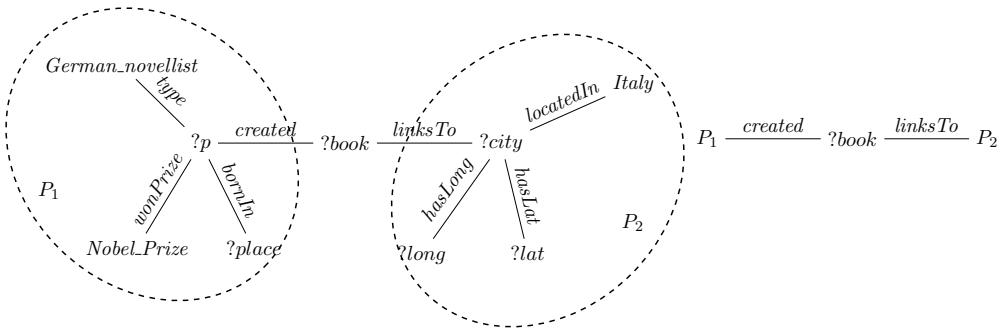
```
   Input: SPARQL graph Q^R
   Result: O – Join Ordering for Q^R
1  begin
2      stars ← getStars(Q^R)
3      DPTable ← []          ▷ stores covered nodes, corresponding partial plan and its cost
4      foreach s ∈ stars do
5          add meta-node for s to Q^R
6          remove joins covering s from Q^R
7          p ← getStarJoinOrder(s)          ▷ see Algorithm 11
8          DPTable.append(s, p, cost(p))
9      add edges between adjacent meta-nodes
10     estimate selectivities using Char.Pairs
11     run DP algorithm on DPTable

12     GETSTARS(Q^R)
13     stars ← ∅
14     order triple patterns in Q^R by subject
15     foreach distinct subject s ∈ Q^R do
16         star ← triples with subject s
17         card ← getCardinality(star)          ▷ using Char.Sets
18         if card < budget then
19             stars.add(star)
20             remove edges within star from Q^R
21     order triple patterns Q^R by object
22     foreach distinct object o ∈ Q^R do
23         star ← triples with object o
24         ...▷ similar to lines 17-20
```

and P_2 in Figure 4.5a. Note that we don't form a star query around the variable `?book`, although syntactically it is a star with one edge. The reason is, based on our characteristic set estimation, we see that this star would return a lot of intermediate results (in fact, all the triples with the `linksTo` predicate). The algorithm stays conservative and does not "oversimplify" the query, leaving more choices to the later Dynamic Programming stage.

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

The algorithm then considers all the stars formed by the GETSTARS subroutine;



(a) SPARQL graph

(b) Simplified SPARQL graph

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

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

Figure 4.5.: Query Graph Decomposition

for every star it adds the new meta-node to the query graph and removes the intra-star edges (lines 5-6). The plan for the star subquery is computed using the Hierarchical Characterisation (see Algorithm 11) and added to the DP table along with the meta-node (lines 7-8).

After all the star subqueries have been optimized, we add the edges between meta-nodes to the query graph, if the original graph had edges between the corresponding star sub-queries (line 9). The selectivities associated with these edges are computed using the Characteristic Pairs synopsis, and the regular Dynamic Programming algorithm starts working on this simplified graph (lines 10-11).

An example of the simplified query graph is given in Figure 4.5b. There, two meta-nodes P_1 and P_2 are connected with the chain. Although the `?book` variable did not form the star subquery, we can still use its emitting predicate `linksTo` to estimate the cardinality of the remaining joins in the query. For instance, in order to estimate the size of the join of P_1 and `?book`, we look up all Characteristic Pairs such that the first set in the pair has predicates from P_1 (`type`, `wonPrize`, `bornIn` and `wrote`), and the second contains the `linksTo` predicate.

Finally, the first four entries of the DP table are depicted in Figure 4.5c. The

optimal plan for the book entity is an index scan, at the same time for P_1 and P_2 we have plans for the corresponding star queries. Since the complexity of DP grows exponentially, even a small reduction of the query graph can greatly improve the runtime of the overall join ordering strategy. In our case, simplifying the query graph from 8 nodes to 3 nodes gives a reduction from $8! = 40320$ plans to $3! = 6$ plans.

4.4. Experiments

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

Algorithms and the Runtime System

To study the performance of our algorithm, we implemented it in the RDF-3X system [78], and compared it with original RDF-3X optimizer (which in turn we studied in two variants: with and without Characteristic Sets [76], denoted DP and DP-CS). In addition, we implemented in RDF-3X the greedy heuristics of [29] (Greedy), and the heuristics SPARQL planner (HSP) [100].
All join ordering algorithms have access to full stack of RDF-3X indexes (including all six permutations of S, P and O, and aggregated indexes). All produced plans are run with the same runtime settings (including the Sideways Information Passing [77]), so the difference between different running times is solely due to the quality of different optimizers. For all the plans, we disable the dictionary ID mapping, since its runtime depends only on the result size (and is the same across all algorithms).
Our prototype is run on a server with two quad-core Intel Xeon CPUs (2.93GHz) and with 64GB of main memory using Redhat Enterprise Linux 5.4.

Datasets and Workload Generator

We used three large Linked Data graphs: YAGO2 [45] with over 110 million triples, LibraryThing [78] with 36 million triples and UniProt [102] consisting of around 850 million triples.
In order to test the scalability and robustness of the query optimization algorithms, we generate the query workload for these three datasets. The generated queries are of two kinds, the star-shaped queries, and the arbitrary (complex) queries.

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

As a result of the node selection procedure we are left with the handful of "central" nodes (from few hundreds in YAGO2 to tens of thousands in LibraryThing). We then form the star-shaped queries around a subset of these nodes by randomly choosing the attributes of these nodes (i.e., the objects connected to it) and replacing some of them with variables. At the end we also replace the central node with the variable itself, so the query gets a form of a natural question "find all the entities with given attribute values, and extract some of their attributes".

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

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

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

Methodology

For every query and every optimization strategy we run the optimizer under test 11 times, and record an average of all runs except the first one (this way we bring the system to the warm cache state). We call this number a *planning time* of the query; we excluded from it the (common for all optimizers) parsing, string-to-ID mapping and code generation time for SPARQL queries.
In theory, the Dynamic Programming strategy is supposed to yield the optimal plan, so we could just record how much slower the other techniques are (including ours, which is also a heuristics for general-shaped queries). But in reality the DP strategy sometimes fails to find the best solution, because of cardinality misestimations during the cost function computation. Besides, sometimes the plan with the best cost function value does not yield the best runtime (although these two quantities are highly correlated).
Therefore, to measure the quality of output plans for different query optimizers, we define the *fitness function* $F(A_i, q)$ of the join ordering algorithm A_i for the given query q as a proportion of the runtime of its output $plan(A_i, q)$ to the best runtime for that query among all algorithms A_i:

$$F(A_i, q) = \frac{\text{runtime}(plan(A_i, q))}{\min_i \text{runtime}(plan(A_i, q))}, \tag{F1}$$

where runtime$(plan(A_i, q))$ is in turn the smallest among 10 warm cache runs of the $plan(A_i, q)$ in the system. For the group of queries (e.g., all star queries of size 8), we report the geometric mean of $F(A_i, q)$ taken accross all queries q_i in that group for each algorithm separately. This indicates the "average quality" of the query optimization strategy for the given group of queries (ranging from 1 to infinity, lower values are better).

Computing Statistics

First we measure the time and space needed to store our hierarchical characterisation – the data structure used for the star queries optimization (and subsequently for optimizing arbitrary queries). Table 4.1 presents the numbers observed while loading the three datasets in RDF-3X. As we see, the Hierarchical Characterisation has the smallest footprint in UniProt dataset. The reason for this small size is probably the well-structured nature of UniProt which basically has a regular schema. This is the ideal case from the indexing standpoint, and the HC works very well in such a setting. The other two datasets are less regular, with more different entities types, but still the overall impact on the database size and loading time is rather small.

Dataset	Space		Time	
	Mb	% of Total Size	s	% of Total Loading Time
YAGO2	5	0.1%	91	3%
UniProt	0.6	0.0001%	10	0.001%
LibraryThing	15	0.5%	62	4%

Table 4.1.: Hierarchical Characterisation: Indexing Space and Time

Dataset	Space		Time	
	Mb	% of Total Size	s	% of Total Loading Time
YAGO2	0.6	0.01%	1.2	0.008%
UniProt	0.01	0.00001%	0.7	0.004%
LibraryThing	0.18	0.0003%	1.5	4%

Table 4.2.: Characteristic Pairs: Indexing Space and Time

Similarly, the time and space needed to index and store all the frequent Characteristic Pairs is very modest comparing to the overall loading time and the database size. As Table 4.2 shows, the extreme case here is again represented by UniProt, which is a very structured RDF dataset. YAGO2 and LibraryThing are somehow less structured, but still the amount of additional information is less than 0.01% of the overall database size, and the indexing time is practically negligible.

Query Optimization: Star Queries

We start with the summary of our experiments with star query optimization. First of all, Table 4.4 reports the total running time (i.e., optimization and plan execution time) for 300 randomly generated star queries per dataset. The queries are grouped by size, from small (5-6 joins) to large (9-10 joins). The reported time is the sum of the join ordering time and plan execution time, with join ordering time also being reported separately in the brackets. We see that, in terms of the overall runtime, our algorithm outperforms even the exact DP and DP-CS strategies. When it comes to just the query optimization time, the clear winner is the HSP planner with all compile times being much less than 1 ms. However, it pays a high price for that speed, since most of the time the generated plans are far from optimal. The Greedy strategy and our query decomposition

take comparable amount of time, while the DP and DP-CS are the slowest, as expected. Note that the significant difference between planning times of DP and DP-CS is due to usage of Characteristic Sets for cardinality estimations. Indeed, for any particular query the CS-based cardinality estimation is rather fast, but it has to be performed for every subquery of the query graph. This renders usage of Characteristic Sets in the standard DP algorithm unfeasible for large query graphs.

The values of the fitness function $F(\mathsf{Algo}, q)$ for different algorithms are given in Table 4.3. Somewhat surprisingly, our algorithm actually produces better plans than the basic Dynamic Programming strategy even without considering the join ordering time. This should be attributed to misestimations of cardinalities (and therefore the cost function) that the DP strategy makes during the join order construction. Note that DP-CS produces the best query plans here due to its accurate (and expensive, as we have discussed above) cardinality estimation procedure based on Characteristic Sets alone. We also found that while for small queries (5 and 6 triple patterns), especially if they have constants, the performance of HSP is comparable with the rest of the algorithms, the large queries can get unpredictably bad plans from the HSP planner. The reason lies in the fact that HSP ignores all the available statistics and makes "blind" decisions that lead to extremely suboptimal plans.

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

	Query Size (number of joins)								
	YAGO2			UniProt			LibraryThing		
Algo	[5,6]	[7,8]	[9,10]	[5,6]	[7,8]	[9,10]	[5,6]	[7,8]	[9,10]
DP	1.20	1.24	1.31	1.19	1.21	1.24	1.18	1.17	1.20
DP-CS	**1.15**	**1.13**	**1.19**	**1.18**	**1.16**	**1.18**	**1.13**	**1.13**	**1.15**
Greedy	1.34	1.38	1.31	1.21	1.33	1.35	1.19	1.21	1.20
HSP	1.22	1.30	2.31	1.22	1.39	3.80	1.18	1.19	2.73
Our	**1.15**	1.18	1.20	**1.18**	1.17	1.19	1.15	1.17	1.18

Algorithms: Dynamic Programming (DP), DP with Char.Sets (DP-CS), Greedy Operator Ordering (Greedy), Heuristic SPARQL Planner (HSP), Algorithm 11 on p.95 (Our)

Table 4.3.: Fitness function $F(\mathsf{Algo}, q)$ (Formula F1 on page 105) for five different algorithms, average over 300 random star queries

Dataset	Query Size (# joins)	*Query Runtime (including Optimization Time), ms*				
		DP	DP-CS	Greedy	HSP	Our
YAGO2	[5, 6]	92 (2)	103 (17)	98 (1)	90 (0.2)	**87 (1)**
	[7, 8]	129 (2)	170 (57)	142 (4)	130 (0.2)	**120 (2)**
	[9, 10]	241 (45)	393 (215)	186 (8)	345 (0.3)	**184 (4)**
UniProt	[5, 6]	94 (2)	110 (18)	94 (1)	95 (0.2)	**91 (1)**
	[7, 8]	142 (8)	186 (58)	151 (4)	157 (0.2)	**132 (3)**
	[9, 10]	257 (45)	421 (220)	239 (9)	703 (0.3)	**207 (4)**
Library-Thing	[5, 6]	95 (2)	105 (16)	95 (1)	93 (0.2)	**91 (1)**
	[7, 8]	140 (8)	188 (61)	140 (4)	**134 (0.2)**	135 (3)
	[9, 10]	266 (44)	404 (208)	230 (8)	499 (0.3)	**222 (4)**

Algorithms: Dynamic Programming (DP), DP with Char.Sets (DP-CS), Greedy Operator Ordering (Greedy), Heuristic SPARQL Planner (HSP), Algorithm 11 on p.95 (Our)

Table 4.4.: Total execution time for star queries, average over 300 random queries per dataset

Query Optimization: General Queries

In Table 4.5 we present the total execution times for queries of the general shape over three datasets. As for star queries, we report the *average* execution time over a large number of *randomly generated* queries per dataset (400 in this experiment). The queries (and corresponding runtimes) are grouped by the query size: from small (10 to 20 joins) to large (40 to 50 joins). Again, the reported time is a sum of both optimization and execution time; the optimization time is also given separately in brackets after the total time. The exact algorithms, DP and DP-CS did not scale for the queries with more than 20 triple patterns (DP-CS takes more than 60s on average for the queries with 10 to 20 joins, so we exclude it from this experiment).

The total execution time of our algorithm is consistently the best among all the competitors: it outperforms the exact DP algorithm for small queries due to more efficient search space enumeration, and gives better approximate solutions to the join ordering problem than any other heuristics (Greedy and HSP). Additionally, unlike the Greedy algorithm, the proposed algorithm scales well in the size of the query. Note that here again HSP is the fastest algorithm to get a query plan, although the quality of its output is the worst among all competitors, so the total execution time for plans yielded by the HSP planner is the largest among all the heuristics, and it quickly gets unacceptably bad as the query size grows. In order to study the quality of output plans in more detail, in Table 4.6 we report the observed values of the fitness function $F(\mathsf{Algo}, q)$ for all the considered algorithms. The plans produced by our algorithm outperform the

Dataset	Query Size (# joins)	*Query Runtime (including Optimization Time), ms*							
		DP		Greedy		HSP		Our	
YAGO2	[10, 20)	7745	(7130)	857	(133)	1025	(2)	**660**	(150)
	[20, 30)	—		1236	(413)	3189	(3)	**967**	(315)
	[30, 40)	—		2204	(838)	4102	(4)	**1211**	(348)
	[40, 50)	—		4145	(1194)	10720	(5)	**2174**	(890)
UniProt	[10, 20)	9823	(9323)	739	(155)	1160	(2)	**566**	(153)
	[20, 30)	—		1220	(422)	2094	(3)	**838**	(330)
	[30, 40)	—		2092	(927)	1952	(4)	**1356**	(701)
	[40, 50)	—		2840	(1180)	7228	(5)	**1755**	(820)
Library-Thing	[10, 20)	8603	(7809)	912	(142)	2187	(2)	**742**	(148)
	[20, 30)	—		1615	(414)	3852	(3)	**1105**	(302)
	[30, 40)	—		2644	(918)	9415	(4)	**1656**	(697)
	[40, 50)	—		3885	(1201)	8970	(5)	**2177**	(813)

Algorithms: Dynamic Programming (DP), Greedy Operator Ordering (Greedy), Heuristic SPARQL Planner (HSP), Algorithm 12 on p.101 (Our)

Table 4.5.: Total execution time of general queries, average over 400 random queries per dataset

Dataset	Query Size (# joins)	DP	Greedy	HSP	Our
YAGO2	[10, 20)	1.81	2.13	3.01	**1.50**
	[20, 30)	—	1.83	7.08	**1.45**
	[30, 40)	—	1.98	5.94	**1.25**
	[40, 50)	—	3.17	11.51	**1.38**
UniProt	[10, 20)	1.43	1.67	3.31	**1.18**
	[20, 30)	—	1.90	4.98	**1.21**
	[30, 40)	—	2.01	3.36	**1.13**
	[40, 50)	—	2.05	8.91	**1.15**
LibraryThing	[10, 20)	1.93	1.88	5.33	**1.45**
	[20, 30)	—	1.96	6.28	**1.31**
	[30, 40)	—	2.15	11.72	**1.19**
	[40, 50)	—	2.44	8.15	**1.24**

Algorithms: Dynamic Programming (DP), Greedy Operator Ordering (Greedy), Heuristic SPARQL Planner (HSP), Algorithm 12 on p.101 (Our)

Table 4.6.: Fitness function $F(\mathsf{Algo}, q)$ (Formula F1 on page 105) for general-shaped queries, average over 400 random queries per dataset

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

Table 4.7.: Evaluation of individual techniques: total runtime of 400 random queries over YAGO2 dataset, ms

competitors almost for all queries. For smaller queries (less than 20 triple patterns) it approaches the quality of the exact DP-based algorithms, and frequently outperforms them, since our Characteristic Pairs data structure captures correlations that are not available to the DP-based algorithms. For larger queries, it is up to 11 times better than the HSP algorithm. Note that using Characteristic Pairs with DP or DP-CS is not really feasible, since it would increase the query compile time of these strategies even more: the Pairs would have to be used to estimate the cardinality for every subplan under consideration, as opposed to our strategy to use it only between certain star-shaped "blocks" of the query.

Effect of Individual Techniques

Finally, we quantify the effect of two techniques presented in the chapter when they are used individually. Namely, we consider a query optimizer that (i) uses only Hierarchical Characterisation (Only HC); (ii) uses only Characteristic Pairs (Only CP); and (iii) uses both of them (HC + CP). Table 4.7 reports the total execution time of 400 random queries of arbitrary shape over the YAGO2 dataset for these three setups.

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

4.5. Related Work

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

The original RDF-3X [78] employs the exact Dynamic Programming algorithm, which, as we have seen, faces computational problems when the query size grows. Its variant, DP with Characteristic Sets, uses more accurate selectivity estimations at the expense of even slower query compile time. The Jena optimizer [95] relies on the greedy join ordering heuristical algorithm, but it tends to underestimate intermediate result sizes [76], which can degrade the query execution time by orders of magnitude. A variant of the greedy algorithm that operates on the data flow graph of the query and takes into account different SPARQL constructs (like OPTIONAL and UNION) is proposed in [17].

Presented approach differs from the original Characteristic Set approach [76] in three main points: first, we extend the CS to the hierarchy and propose a linear-time join ordering heuristics based on the Hierarchical Characterisation; second, we introduce a novel statistical structure to capture the "foreign-key"-like relationships in RDF graph; finally, we suggest the query simplification algorithm for general SPARQL queries. Apart from the cardinality estimation problem, the Characteristic Sets can be used as a foundation for the physical layout of the RDF store [70].

The recently proposed Heuristical SPARQL Planner [100] introduces several heuristical rules of ordering the joins based merely on the structure of the SPARQL query. However, completely ignoring the data statistics leads to unpredictably bad query plans, as we have demonstrated in out evaluation.

Another approach towards SPARQL optimization is to translate the SPARQL query to its SQL equivalent, and then optimize this intermediate SQL query [28]. This, however, prevents the optimizer from using RDF-specific information about correlations between predicates in star subqueries and chains. Besides, a straighforward translation of all SPARQL joins to SQL joins leaves the relational optimizer with the search space of enormous size.

Our query simplification techniques are inspired by the query simplification mechanism for SQL queries [75]. We also reduce the search space size by making some simplification before the DP algorithm starts. However, our simplification is driven by the query structure: namely, we are using the SPARQL-specific star-subqueries as building blocks for the DP algorithm.

CHAPTER 5

Query Optimization for Property Graphs

In this chapter we turn our attention to query optimization problems for Property Graphs, an emerging graph representation model, and Cypher, the only declarative graph query language implemented in an available database system. Cypher is a query language supported by the open-source Neo4j database system, and here we describe the query optimizer for that system. Although general ideas of query optimization for Cypher are quite similar to the ones already implemented for SPARQL in RDF-3X, it is still interesting to consider their application in another data model. Specifically, this chapter discusses the following technical issues:

- We propose a new algebra for Cypher that matches both the graph pattern matching capabilities of the language and the features of the runtime system. In particular, we will prefer expansions over edges (as opposed to index scans and merge joins in RDF-3X) as a basic operator in the language.
- Within this algebra we present a customized version of the greedy heuristics that yields an approximation to the optimal logical plan for a query in polynomial time.
- We discuss how one can estimate cardinalities of subqueries in Cypher, employing some basic statistics provided by the storage engine.
- We compare the query optimizer based on the principles described in the chapter with the old query processor of Neo4j in a comprehensive experimental study involving three different datasets and workloads of different complexity.

The work presented in this chapter is a result of a collaboration with Neo Technology, the company behind the popular open-source Neo4j graph database[1]. All of the described concepts and algorithms were prototyped and/or implemented

[1]The author thanks the Neo Techology's team, in particular Andres Taylor, Jakub Wieczorek, Stefan Plantikow, Davide Grohmann and Alex Averbuch

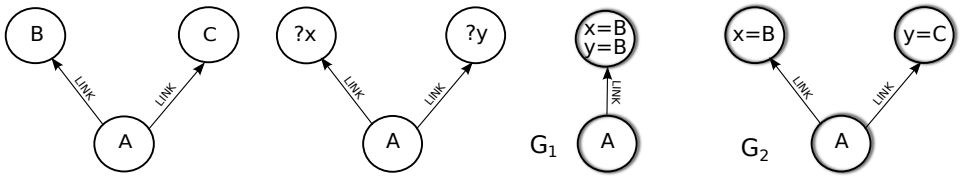
in the Neo4j database system. The experiments were conducted on a prototype query optimizer implemented on top of the Neo4j's runtime system.
The chapter is structured as follows. We begin with pointing out the difference in graph pattern matching in SPARQL and Cypher. We will then describe the algebra for Cypher in Section 5.2, and then turn to the two main problems of query optimization: ordering of logical steps and cardinality estimation (Section 5.3). We conclude with description of experiments.

5.1. Graph Pattern Matching in SPARQL and Cypher

Both SPARQL and Cypher, albeit very different in syntax, essentially allow to formulate graph pattern matching queries over graphs in the RDF and Property Graph data model, respectively. There is, however, a subtle yet important difference in the types of matching for the two languages.
To describe this difference in a formal way, we would need a formal definition of Cypher's semantics, which is outside of the scope of this thesis. Instead, we will illustrate the essence of the issue with a simple example. Consider a small graph depicted in Figure 5.1a, and the two (seemingly) similar queries, in SPARQL and in Cypher, that extract neighbors `x` and `y` of a given node `A`. The graphical representation of both queries is given in Figure 5.1b.

```
select ?x ?y
where {
    A link ?x.
    A link ?y. }
```

```
MATCH (A)-[:LINK]->(x),
      (A)-[:LINK]->(y)
RETURN x, y
```



(a) Data graph **(b)** Query pattern **(c)** Answer subgraphs

Figure 5.1.: Graph homomorphism vs Graph isomorphism in pattern matching queries

The SPARQL query returns, among other answers, the subgraphs G_1 and G_2 shown in Figure 5.1c, which are both *homomorphic* to the original graph G in a sence that a pair of connected nodes in G_i can be mapped to a connected pair in G. However, graph homomorphism allows two distinct nodes of G to be mapped to a single node in G_1, thus changing the shape of the graph. This is not the case for *graph isomorphism*, where the mapping has to be a bijection. For example,

G_2 is isomorphic to G, since any two distinct nodes from G are mapped to two distinct nodes from G_2. Cypher query semantics, unlike SPARQL's, operates in terms of graph isomorphism, therefore the Cypher query only returns G_2 as an answer.
From the query processing point of view this means that the Cypher query processor needs to make sure that different variables are mapped to distinct nodes, and a straightforward way to do so is to add the selection $x \neq y$ on top of the query plan.

5.2. Cypher Algebra

We start with defining an algebra for Cypher. Our goal is to have a set of algebraic operations that (i) closely corresponds to the Cypher's constructs, such as matching nodes and edges based on various conditions, and (ii) matches Neo4j's runtime system, in that the edges of the graph can be traversed ("followed") efficiently.

Basic operators

At the foundation of the algebra there is an operator that yields all nodes in the graph, $\mathcal{N}$. The rest of the operators is defined recursively. Let $\mathcal{F}(P)$ denote the set of node and edge variables in the algebraic expression P.
On top of an expression P we can define selections that filter out nodes with specified properties (or labels): operator $\sigma_f(P)$ where f is a boolean expression involving variables from $\mathcal{F}(P)$. Physical implementations of selections naturally differentiate between selection by node ID (in this case we apply `NodeIdSeek`), access to nodes of a particular label (the selection is implemented with `NodeLabelScan`), and selection of nodes with specified indexed properties (realized with the `NodeIndexSeek` operator). From the algebraic point of view, of course, all of these methods are not distinguishable.
In order to match edges, we introduce a unary `Expand` operator $\mathcal{E}^l_{a\to b}(P)$. It describes all nodes `b` that are reachable via one outgoing edge with the given label l from any node `a` (whose bindings are defined by the expression P, in other words $\texttt{a} \in \mathcal{F}(P)$). The operator introduces new variable `b`, and we require that $\texttt{b} \notin \mathcal{F}(P)$. To simplify our notation, we will omit the label l when it can be deduced from the context. `Expand` can also produce bindings for edges, if the edge label is not given (i.e., for query constructs of a form `(n)-[e]-(m)`, where `e` is a variable). To denote expansion on incoming edges, we write $\mathcal{E}_{a\leftarrow b}(P)$.
Note that fundamentally this is just a way of expressing a join between the source nodes `a` and the entire graph; the join condition is `a` reaches `b` *via one*

edge. We found it more intuitive to model it as a specialized operator, since it fits both the Cypher syntax, where paths are first-class citizens, and the Neo4j storage system, where following one-hop paths from a given set of nodes can be done extremely efficiently.
Additionally, we use traditional relational operators like joins, outer joins and aggregates. For the two joined expressions P_1 and P_2, the join conditions between them may be specified both in terms of node bindings (i.e., certain node bindings should be the same in both expressions), or in terms of properties of certain nodes (e.g., $v.p1 < w.p2$ for $v \in \mathcal{F}(P_1)$ and $w \in \mathcal{F}(P_2)$)
These operators are already enough to translate a substantial portion of Cypher's pattern matching queries into their algebraic representations. Consider the query given in Figure 5.2a. This query performs a triangle matching, where the type of one of the nodes (identified by `b`) is set to `:T`. As usual with declarative queries, there are multiple equivalent algebraic representations, so we depict two of them in Figure 5.2b. One way to answer this triangle query is to keep expanding from nodes `b` to nodes `a` and `c`, and then find all $\mathtt{a}'$ reachable from c. Final selection is necessary to "close" the triangle at `a`. Another possible way is to break this chain of `Expand`s into two independent chains of expansions that are later joined. Note that even in this tiny example there are two ways of splitting the chain of `Expands` and placing a join, after $\mathcal{E}_{b \leftarrow a}$ (shown in Figure 5.2b) or after $\mathcal{E}_{b \rightarrow c}$. This, of course, hints at the exponential search space — a typical feature of query optimization problems.
In the example above we have implicitly used certain algebraic properties of the `Expand` operator. Let us now formulate them.
First, two `Expand` operators can be re-ordered as long as they are both defined:

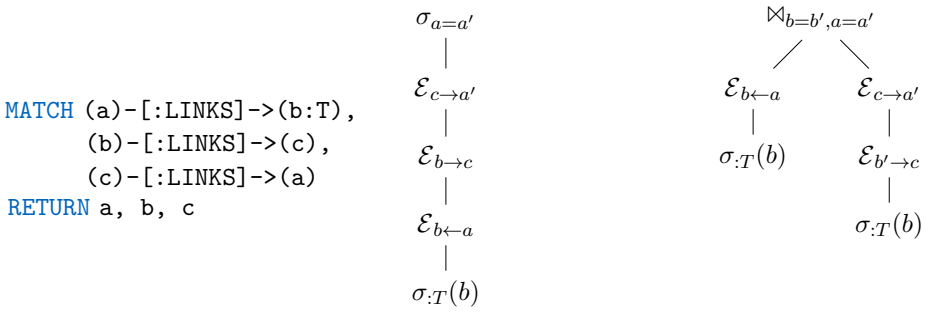
$$\mathcal{E}_{b \rightarrow a}(\mathcal{E}_{b \rightarrow c}(P)) = \mathcal{E}_{b \rightarrow c}(\mathcal{E}_{b \rightarrow a}(P)) \quad \text{with} \quad b \in \mathcal{F}(P), \tag{E1}$$

where directions of edges do not matter.
Second, an `Expand` and a join are also reorderable:

$$\mathcal{E}_{b \rightarrow a}(P_1 \bowtie P_2) = \mathcal{E}_{b \rightarrow a}(P_1) \bowtie P_2 \quad \text{with} \quad b \in \mathcal{F}(P_1),\, b \notin \mathcal{F}(P_2), \tag{E2}$$

$$\mathcal{E}_{b \rightarrow a}(P_1 \bowtie P_2) = \mathcal{E}_{b \rightarrow a}(P_1) \bowtie \mathcal{E}_{b \rightarrow a}(P_2) \quad \text{with} \quad b \in \mathcal{F}(P_1),\, b \in \mathcal{F}(P_2) \tag{E3}$$

Note that, as with relational joins, all the constraints on the variables (such as $b \in \mathcal{F}(P_1)$) are automatically guaranteed by the shape of the query graph.
Property (E3) also holds for the left outer join $\bowtie_p$ if the join predicate p *rejects null values*, that is if p evaluates to false whenever any of its free variables is set to `NULL`.



(a) Cypher query **(b)** Two equivalent algebraic representations

Figure 5.2.: Cypher query and its two algebraic expressions

These properties of our algebraic operators mean that we can directly apply the Dynamic Programming algorithm from Chapter 1, or any bottom-up plan construction heuristics, to find the optimal ordering of operators for a Cypher query.

Advanced operators

There are a few other Cypher constructs that require additional algebraic operators. For completeness, we briefly describe them in this section.

OPTIONAL match. Similarly to SPARQL, it is also possible to mark some patterns as optional in the query. If there are no nodes or edges that satisfy the optional pattern, `NULL` values will be used as the corresponding bindings. The following query illustrates this concept:

```
MATCH (a:Label1)
OPTIONAL MATCH (a)-[:LINKS]->(b)
RETURN a, b
```

The query will return all nodes `a` with the given label `Label1`, and all their neighbors `b` (connected via `LINKS`) or `NULL`, if there is no such neighbor. This behavior corresponds to the relational outer join operator.

Shortest paths. Cypher allows to explicitly address the shortest path between two nodes by using `shortestPath` function in the query text, as well as to list all shortest paths between a given pair of nodes with `allShortestPaths`. Paths are also the first class citizens of the language, in particular a query can formulate conditions on the path length or nodes along the path. Algebraic handling of the `shortestPath` operator (including cardinality estimates and operator ordering)

is similar to the `DijkstraScan` operator, discussed in Chapter 2.

Subqueries. A Cypher query can have nested subqueries, where some of the bindings defined in the outer query are passed to the nested `MATCH` clauses. Take Query 5.1 as an example, where bindings of `p` (path) and `b` (node) are passed to the second `MATCH` clause, where we match the node `c` and filter out some nodes `b` based on their property values.

```
MATCH p = (a:Label)<-[:LINKS]-(b)
WITH p, b
MATCH (b)-[:LINKS]->(c)
WHERE b.x > 0
RETURN p, b, c
```

Query 5.1: Cypher query with a subquery

Subqueries are a convenient mechanism of splitting graph pattern into several parts that allows the user to manipulate the results of the first part (by computing aggregations, ordering and limiting the number of result entries etc.) before passing it to the further `MATCH` clauses.

As we have seen in the example above, the semantics of Cypher subqueries is similar to that of SQL subqueries: for every node binding from the first query in the chain we need to evaluate all the further queries (recursively apply the same rule to the rest of the chain). In order to reason about subqueries on the algebraic level, we use the `Apply` operator $\mathcal{A}$, introduced in [30]. `Apply` gets two expressions as input, P and $E(r)$ (with $r \in \mathcal{F}(P)$). The operator evaluates the expression $E(r)$ for each value of r. Note that we have slightly modified the original $\mathcal{A}$ definition: in [30] it takes a relational table as a source of parameter bindings, whereas we allow arbitrary expressions P as binding sources. Since the second input of our `Apply` is a parametrized expression $E(r)$ with r as a parameter, we will use (r) as a leaf expression in the right sub-expression of `Apply`.

There is a physical operator that executes $\mathcal{A}$ in the Cypher's runtime system, which is essentially a form of nested-loop join. In order to enable more powerful join techniques, `Apply` has to be unnested, using the following algebraic equivalences (we omit the parameter in the right side when it is not important):

$$L \; \mathcal{A} \; (r) = L \tag{A1}$$

$$L \; \mathcal{A} \; ((r) \; \mathcal{A} \; R) = L \; \mathcal{A} \; R \tag{A2}$$

$$L \mathcal{A} \sigma_f(R) = \sigma_f(L \mathcal{A} R) \tag{A3}$$
$$L \mathcal{A} \mathcal{E}_{a \to b}(R) = \mathcal{E}_{a \to b}(L \mathcal{A} R) \tag{A4}$$

Similar to [30], we push down Apply until the right child does not have the parameter whose bindings are supplied by the left side. Note that un-nesting does not introduce equi-joins, but rather relies on `Expands`: our goal is to construct *some* equivalent algebraic expression without $\mathcal{A}$, this way joins can be later considered during the logical plan construction (operator ordering procedure). As an example, consider Figure 5.3 which depicts an initial expression with `Apply` for Query 5.1 and how it can be un-nested.

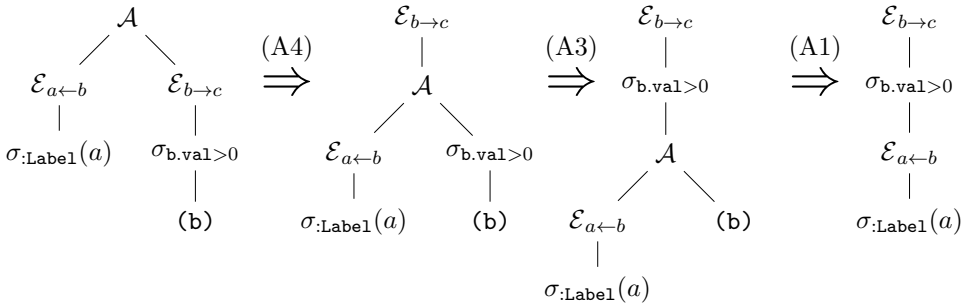


Figure 5.3.: Unnesting of the `Apply` operator. Arrows are annotated with the rule number

Pattern predicates. Along with predicates on properties, mapped to the regular selection operator, it is also possible to specify existential predicates on path patterns. In other words, when the pattern of a form `(node1)-[:edge]-(node2)` occurs in the `WHERE` clause of the query, it becomes a predicate that checks the existence of the corresponding pattern in the graph, without creating new bindings for the nodes. In this case, of course, bindings for at least one of `node1` or `node2` have to come from outside of the pattern predicate. The negation on the path predicate denotes the absence of the path.
Pattern predicates therefore resemble relational EXISTS/NOT EXISTS subqueries, and we represent them with `SemiApply` and `AntiSemiApply`, respectively. Naturally, the path pattern may have more than one edge (and therefore more than one `Expand` in the algebraic expression), so it is a non-trivial subquery, not just one operator. When unnested, `SemiApply` and `AntiSemiApply` correspond to the semi- and anti-semi-join. Full unnesting of all types of $\mathcal{A}$ (together with aggregation) is subject of future work.

5.3. Query Optimization

Since we have already discussed the general architecture and specific problems of a cost-based query optimizer for graph databases in Chapter 1 and Chapter 4, we will only briefly overview the query optimization for Cypher, highlighting the parts that differ from RDF-3X. This general overview is presented in Section 5.3.1. We will then dive into details of the algorithm for the logical plan construction, i.e. join ordering, in Section 5.3.2, and discuss the cost model and cardinality estimations in Section 5.3.3.

5.3.1. General Architecture

Given a query, the Cypher optimizer performs the following steps:

1. Turns it into an Abstract Syntax Tree (AST), checks semantics (e.g., that all labels exist), normalizes it (collects together different path matches and predicates)

2. Builds a query graph representation of a query. The graphical representation of a query is essentially a query graph (cf. Figure 5.1b). Unlike SPARQL, however, edges in the query graph do not correspond to joins, but rather to expansions. As we have discussed in the previous section, two plans P and Q can be joined iff $\mathcal{F}(P) \cap \mathcal{F}(Q) \neq \emptyset$, or – in terms of the query graph – if the corresponding subgraphs share at least one node.

3. Deals with subqueries and then finds the optimal operator order (for joins, `Expands` and `Apply` operators that may be left after partial unnesting). We will describe the algorithm for that in Section 5.3.2.

4. Translates the optimal plan into the database API methods for data access.

Note that the original version of Neo4j's query processor (i.e., prior to version 2.0) followed a similar logic of query execution, except that step 4 was performed by following several "hard-coded" transformation rules that were meant to translate AST into imperative plan. We will quantify the difference between the cost-based optimizer and the rule-based one in Section 5.4.

5.3.2. Operator Ordering

In this section we describe a greedy algorithm that constructs a logical plan for the query graph in a bottom-up manner. Constructed plans are kept in a `PlanTable` whose structure is quite similar to the table employed by the Dynamic Programming algorithm. Here, an entry of `PlanTable` contains a

Algorithm 13: Greedy operator ordering for Cypher

```
Input: Query graph Q
Result: Logical plan P that covers all nodes from Q
1  P ← [ ]                                                  ▷ PlanTable
2  foreach n ∈ Q do                                         ▷ every node in the query graph
3      T ← constructLeafPlan(n)                             ▷ take selections into account
4      P.insert(T)
5  do
6      Cand ← [ ]                                           ▷ candidate solutions
7      foreach P1 ∈ P do
8          foreach P2 ∈ P do
9              if CanJoin(P1, P2) then
10                 T ←constructJoin(P1, P2)
11                 Cand.insert(T)
12     foreach P1 ∈ P do
13         T ←constructExpand(P1)
14         Cand.insert(T)
15     if Cand.size ≥ 1 then
16         T_best ←pickBest(Cand)                           ▷ pick the plan with the smallest cost
17         foreach T ∈ P do
18             if covers(T_best, T) then
19                 P.erase(T)                               ▷ delete plans covered by T_best
20         T_best ←applySelections(T_best)
21         P.insert(T_best)
22 while Cand.size ≥ 1
23 return P[0]
```

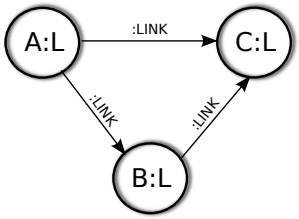
logical plan that covers certain part of the query graph (identified by IDs of nodes in that subgraph), along with the cost of the plan and its cardinality.
As usual, we assume that a query graph is connected; for unconnected graphs we first find the optimal plan for every connected component and then combine them with cross-products.
In order to describe the algorithm, we need to define a specific relationship between a pair of logical plans. We say that a logical plan P (essentially, an algebraic expression) *covers* another logical plan Q, if $\mathcal{F}(Q) \subset \mathcal{F}(P)$, in other words if the plan P covers all the nodes from the plan Q. In terms of a query graph it means that Q contains solution for a subgraph of the query graph of P. Note that Q itself does not have to be a subexpression of P, but rather contain a solution to a subproblem of P.

The algorithm starts off by seeding the `PlanTable` $\mathcal{P}$ with leaf plans that cover individual nodes in the query graph (Algorithm 13, lines 1-4). At every step, we collect the candidate solution that are formed by either joining a pair of plans from the `PlanTable` (lines 7-11) or by expanding a single plan via one of the edges of the query graph (lines 12-14). Note that the actual implementation also considers shortest paths and aggregates that are left out in the pseudocode. The algorithm then picks the candidate plan with the best cost, and deletes all the plans from $\mathcal{P}$ that are covered by it (lines 16-19). We then apply all syntactically possible selections, and put the plan into $\mathcal{P}$. The procedure is stopped as soon as there are no candidates to consider. At this point, since the query graph is connected, the `PlanTable` has to contain a single plan that covers all the nodes, which we return as a result.

The main loop (lines 5-22) is repeated at most n times (with n being a number of nodes in the query graph), since we start with n plans in $\mathcal{P}$ and remove at least one plan at every step. Then, assuming that `canJoin` utilizes the Union-Find data structure for disjoint sets, the complexity of the entire algorithm becomes $\mathcal{O}(n^3)$.

As an example of the Greedy operator ordering algorithm, consider the following Cypher query with its query graph:

```
MATCH (A:L)-[:LINK]->(B:L),
      (A:L)-[:LINK]->(C:L),
      (B:L)-[:LINK]->(C:L)
RETURN A, B, C
```



We give the step-by-step trace of the algorithm for this query in Figure 5.4, displaying the `PlanTable` together with the candidate list and the best plan T_{best}. In the beginning (`Step 1`), the table is initialized with the plans that provide the best access to single nodes. In our case these are three label scans $\sigma_{:L}$ that extract nodes of the given label `L`. At `Step 1`, the only way to extend these three plans is to form various expansions from the nodes along the edges of the query graph. There are in total 6 expansion possibilities, and based on the cost calculation (not shown in Figure 5.4) we pick T_{best}, eliminate all the plans from $\mathcal{P}$ that are covered by T_{best} (namely, plans for nodes A and B), and proceed to `Step 2`. There again, only expansions from existing plans are possible, with the best candidate being $\mathcal{E}_{B\to C}(\sigma_{:\mathrm{L}}(B))$. This plan replaces the plan for $\{B\}$, but leaves the plan for $\{A, C\}$ since it does not fully cover it.

At `Step 3` we get the first chance to join two subproblems, since now there are

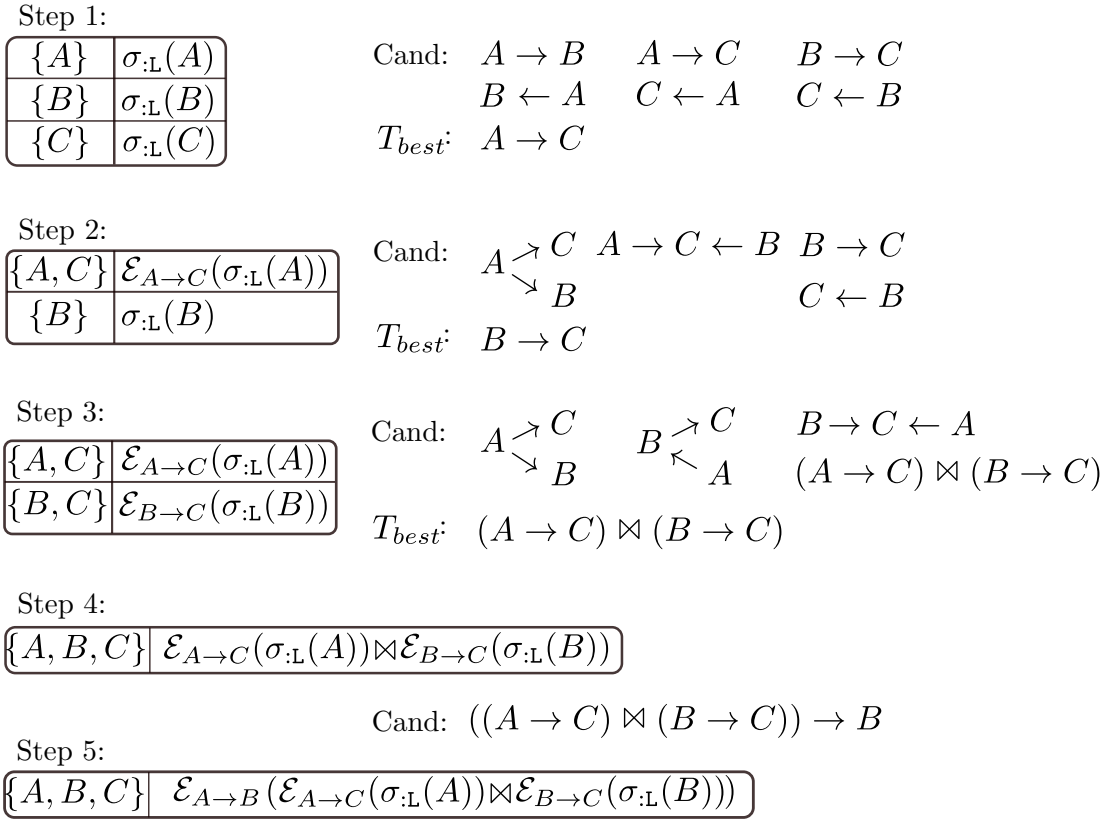


Figure 5.4.: Greedy Operator ordering for Cypher

two plans P and Q in the `PlanTable` with overlapping sets of variables. The resulting plan turns out to be better than any other candidate, and replaces all partial plans in $\mathcal{P}$. At the next step, however, we find out that there is one more possibility to expand from this plan, namely to take the edge $A \to B$, which leads to the plan depicted in $\mathcal{P}$ at `Step 5`. Note that the last `Expand` operator should produce bindings B' and it should be followed by a selection that ensures $B = B'$, but we omit this selection for the sake of brevity.
Naturally, the presented greedy algorithm only yields an approximation to the optimal plan. Future work includes exploiting Dynamic Programming-like algorithm with query graph simplification when needed.

5.3.3. Cardinality Estimation

In this section we discuss cardinality estimation for some simple and frequent types of Cypher subqueries. Our goal is to derive robust estimates based on the statistics extracted from the storage engine.
The engine provides us with with the following base cardinalities (which, in turn,

are coming from the underlying Lucene index):

- Number of nodes with a given label `L`
- Number of nodes with a given label `L` and the fixed value of one of the properties. In other words, we know the exact cardinality of the following (sub)query:

```
MATCH (a:L)
WHERE a.prop = value
```

- Probability distribution of the type of outgoing/incoming edges for nodes with a label `L`, i.e. the result size of the following query (for fixed `L` and `LINK`):

```
MATCH (n1:L)-[:LINK]->(n2)
RETURN n1, n2
```

- Number of edges with a given label `LINK`:

```
MATCH (n1)-[:LINK]-(n2)
RETURN n1, n2
```

Compared to typical RDF engines, the set of these base cardinalities is actually quite rich: since the Property Graph model is less verbose than RDF, it is possible to get as base cardinalities values that would be resulting from many joins and aggregates in SPARQL (e.g., the probability distribution of a given property for a given node type).

Single edge pattern cardinality. We derive the estimate for the important case of edge matching, where the pattern has a general form `(a:X:Y)-[:T1 | T2]->(b:W:Z)`. This matches all the node pairs (where node `a` has both labels `X` and `Y`, and node `b` is labeled with `W` and `Z`) that are connected via an edge with the label `T1` or `T2`. We want to compute an upper bound of the cardinality of the match, rather than rely on the independence assumption leading to severe under-estimation. The number of edges labeled `T1` and starting from nodes with labels `X` and `Y` is bounded by $t_1 = \min(\texttt{X} - [:\texttt{T1}] \rightarrow (), \texttt{Y} - [:\texttt{T1}] \rightarrow ())$, both of the values under min are base cardinalities. Similar formula can be used to get an upper bound t_2 for edges with the `T2` label. Now, $t_1 + t_2$ gives an upper bound to the number of results of the following (sub)query:

```
MATCH (a:X:Y)-[:T1|:T2]->()
```

Using the same argument and the base cardinalities for incoming edges, $t'_1 + t'_2$ bounds the number of matches for

```
MATCH ()-[:T1|:T2]->(b:W:Z)
```

Now, in order to get an upper bound of the entire pattern, it is enough to take $\min(t_1 + t_2, t'_1 + t'_2)$, since the entire pattern is an intersection of the two partial patterns.
Our formula easily generalizes to the case of multiple node and edge labels.
This formula corresponds to estimating cardinality in SPARQL queries with Characteristic Pairs, proposed in Chapter 4. Here, labels (i.e., node types) correspond to Characteristic Sets in RDF (labels are given by the user, as opposed to CS that are discovered by the engine). The cardinality of a connection (a foreign key in relational terminology, a Characteristic Pair from Chapter 4) between two labels is estimated by the optimizer using essentially the same principle. Similar to relational and RDF engines, at some point the Cypher optimizer has to "flee from knowledge to ignorance" [69], assuming independence between types of nodes and edges on one hand and values of properties on the other hand, for instance in queries of a form

```
MATCH (a:L)-[:T1]-(b:M)
WHERE a.prop = value
RETURN a, b
```

Patterns of multiple edges. We consider a pattern of a form `(a:X)-[:T*1..l]->(b:Y)`, where l is a constant. This matches all the paths of length up to l, where start and end nodes have given labels, and all the edges along the path have label `T`. For $l = 1$, this pattern is identical to a simple edge match, considered above. For $l = 2$, we split the pattern into two components:

```
L: (a:X)-[:T]->()
R: ()-[:T]->(b:Y)
```

We know the cardinalities of both `L` and `R`, since they are given by our base statistics. Since there are no restrictions on the middle node, the cardinality of the whole pattern can be estimated as $|\texttt{L}| \cdot |\texttt{R}|$.
If l is larger than 2, however, we again start to "flee to ignorance": the pattern can be split into three parts, `L`, `R` (similar to the case $l = 2$) and `M = ()-[:T*1..l-2]->()`. Since there are no restrictions on nodes from `M`, its cardinality can be estimated as $(l-2)^{deg(:\texttt{T})}$, where $deg(:\texttt{T})$ is an average number of edges with the label `:T` outgoing from a single node (similar to the average node degree, and can be computed from our base statistics). This results in the total estimate of $|\texttt{L}| \cdot |\texttt{M}| \cdot |\texttt{R}|$. This number grows very quickly, and makes the optimizer very cautious about considering the corresponding pattern too early in the plan construction.

Shortest path cardinalities. Estimating the number of shortest path matches in the Property Graph model can in fact be done in the same manner as for RDF graphs (Chapter 2).

5.4. Evaluation

In this section we describe our experiments with the new query optimizer of Neo4j.
The numbers displayed here do not necessarily represent the performance of Neo4j's most recent version. They also do not necessarily represent the best performance achievable with the system, since we have not done any tuning. The only goal that we pursue with this experimental study is to quantify effectiveness of techniques presented in the chapter. For this reason we report on the performance of two versions of the system: `Neo4j-RBO` (with the old rule-based optimizer) and `Neo4j-CBO` (with the new cost-based optimizer, described in this chapter). Both versions share the same query execution layer and storage system, therefore the numbers will indicate strengths and limitations of our new optimizer. All our experiments were run on a EC-2 machine of the type M3.large, with 7.5 Gb of RAM and Intel Xeon 2.6 Ghz processor.
Our first experiment is running simple graph pattern matching queries (up to three edge patterns) over a large music encyclopedia MusicBrainz that contains music metadata. The corresponding graph has 35.7 Million nodes and over 146 Million edges, and occupies 18.85 Gb on disk.
We run 6 queries (their full text is available in Appendix C) and record *Runtime* and *Compile time* for both optimizers, numbers are reported in Table 5.1. We report an average *warm cache* time for every query. We compute the *Improvement* score as a ratio of total runtime of the new optimizer divided by the total time of the old optimizer. The score ranges from 0.85 (this is the query whose total time actually got slower) to 1.8. As we look closer at these numbers, however, we see that the large share of time is spent compiling the query plan for both `Neo4j-RBO` and `Neo4j-CBO`, while the runtime of the plans of `Neo4j-CBO` is up to 12 times better. The fact that compile time dominates the execution time is a typical situation for interactive queries.
We now turn to more complex graph pattern matching queries, this time over the AccessControl graph with 6 Million nodes and 12 Million edges (total size 1.54 Gb). We construct 4 queries with 12 to 18 edge patterns (see Appendix C for their full text), and report their performance (warm cache) in Table 5.2. Now the difference between the old and the new optimizers is quite significant. Note that the compile time has on average increased. This was expected, since the Greedy operator ordering is more time consuming that the simple rule-based

Query	Neo4j-RBO		Neo4j-CBO		Improvement
	Time, ms				
	Runtime	*Compile*	*Runtime*	*Compile*	
Q1	366	1360	31	912	1.8
Q2	11	64	1	63	1.17
Q3	28	288	6	367	0.85
Q4	20	44	6	55	1.04
Q5	83	158	31	113	1.67
Q6	149	513	85	361	1.48

Table 5.1.: Runtime and compile time of basic pattern matching queries over the MusicBrainz dataset

Query	Neo4j-RBO		Neo4j-CBO		Improvement
	Time, ms				
	Runtime	*Compile*	*Runtime*	*Compile*	
Q1	24592	41	1116	195	18.7x
Q2	3191	468	1235	540	2.0x
Q3	3537	454	1019	493	2.6x
Q4	411	54	1	187	2.5x

Table 5.2.: Runtime and compile time of complex pattern matching queries over the Access Control dataset

compilation. This effort, however, clearly pays off, as the runtime numbers for the new optimizer have decreased by up to two orders of magnitude.
Finally, we consider the recently proposed Social Network Benchmark (SNB) by the Linked Data Benchmark Council. We will provide the background information on SNB in Chapter 6, where we describe related benchmarking issues. For now it is sufficient to note that these are 13 queries (see Appendix D) over the synthetic social network dataset (we have generated 1 Gb of data). LDBC queries are extremely challenging for the query optimizer, surpassing the challenges of the classical TPC-H benchmark. This is not an official (audited) LDBC benchmark run, but we follow the spirit of its rules, in that there are no precomputed partial or complete query results. We report the runtime of the queries in Table 5.3. There are four queries that performed much worse with the new optimizer. In all four cases the reason was subqueries that were not unnested and instead executed with the nested loop-style operator. Specifically, the following situations are currently not handled by the prototype, causing very suboptimal performance of the generated plans: complex subqueries with negations, and the combination of aggregation and outer joins in subqueries.

Query	Neo4j-RBO	Neo4j-CBO	Improvement
	Time, ms		
	Runtime	*Runtime*	
Q1	12028	2069	5.8
Q2	1117	486	2.30
Q4	713	695	1.02
Q5	7894	68811	0.11
Q6	485	3690	0.13
Q7	558	179	3.11
Q8	378	197	1.91
Q9	74406	141977	0.52
Q10	8915	1348753	0.006
Q11	484	228	2.12
Q12	14447	210	68.48
Q13	1	1	1

Table 5.3.: Runtime and compile time for SNB LDBC Benchmark, Scale Factor 1

These shortcomings are the subject of future work.

CHAPTER 6

Selecting Parameters For Graph Queries

Parts of this chapter were published in [36]

Throughout the thesis we studied specific techniques that improve query performance for graph databases. In this final chapter we consider a problem of creating a query workload that facilitates comparison of different graph database systems, i.e. a problem of benchmarking for graph technology.

Benchmarking in general raises a number of issues — from data generation to audit — that we will not discuss here. We will focus our attention on a very specific technical problem that helps comparing different query engines (and in particular, different query optimizers), namely: given a specific query template, how do we create a set of parameters such that all the invocation of this query have stable runtime behavior for any cost-based query processor.

In this chapter we show that this requirement of runtime stability becomes a hard challenge when the underlying data features correlations and non-uniform value distribution. We have already seen that these correlations and skewed distributions are commonplace in graph datasets, and in fact our main motivation for this problem is parameter selection for the recently proposed Social Network Benchmark by the Linked Data Benchmark Council. In this chapter we present our solution to the problem, a scalable procedure coined *Parameter Curation* that creates sets of parameter bindings for queries that guarantee runtime stability.

We start with the background information on the Social Network Benchmark and its design principles in Section 6.1. Then, we present examples of anomalies in query runtime behavior caused by data correlation and non-uniform value distributions in Section 6.2. We give a formal problem definition in Section 6.3 and then describe our solution in Section 6.4. Experimental results on queries from the SNB benchmark, presented in Section 6.5, will point out effectiveness and efficiency of our approach.

6.1. Background

In this section we will give a brief overview of the LDBC Social Network Benchmark (SNB); this will facilitate the exposition of advanced benchmarking issues. We concentrate on the principal components of the benchmark: its synthetic dataset, underlying methodology for query design, and the set of queries.

SNB dataset

The underlying dataset is a synthetic social network with rich metadata about users and user activities, based on the S3G2 generator [71]. Its core is the social graph between users, the users have various interests (hashtags), they create content (posts, pictures, comments) that is also annotated with hashtags and locations. We provide a subset of the schema, including all the entities that are used in this chapter in Figure 6.1. Throughout the chapter we are going to use SQL formulation of queries for brevity, so we use a relational schema of the dataset (for example, rows in the `Knows` table in graph databases would become edges with attributes). The main entities are the following:

- `Person`: contains information about names, location, birthday of a user and a creation date of the online profile. Friendship between users is captured in table `Knows`
- `Forum`: represent an online group of people that discuss some topics. Each forum has a moderator, a title, a creation date and a set of tags.
- `Post`: users create `Posts` in `Forums`, which then can receive `Replies` (stored in the same table). `Posts` can also contain `Pictures`.
- `Tag`: denote the topics of posts and comments, as well as interests of users. The actual names of tags are drawn from DBPedia.
- `Organization`: is used for universities and companies. The fact that a user went to a certain university is stored in the corresponding table (the schema for users and companies is exactly the same).

The main property that distinguishes the SNB dataset from other synthetic graph datasets is realistic dependencies (correlations) between properties of the users and the friendship graph structure, for example:

- with high probability friends have studied or worked together

Person
p_personid
p_firstname
p_lastname
p_gender
p_birthday
p_creationdate
p_locationip
p_browserused
p_placeid

Forum
f_forumid
f_title
f_creationdate
f_moderator

Forum_Person
fp_forumid
fp_personid
fp_creationdate

Post
ps_postid
ps_imagefile
ps_creationdate
ps_locationip
ps_browserused
ps_language
ps_content
ps_length
ps_creatorid
ps_locationid
ps_forumid
ps_replyof

Tag
t_tagid
t_name
t_url

Post_Tag
pst_postid
pst_tagid

Person_Tag
pt_personid
pt_tagid

Knows
k_person1id
k_person2id
k_creationdate

Likes
l_personid
l_postid
l_creationdate

Organisation
o_organisationid
o_type
o_name
o_url
o_placeid

Person_University
pu_personid
pu_organisationid
pu_classyear

Figure 6.1.: Part of LDBC schema, used in this chapter

- the interests and geolocations of friends are correlated: users that are connected in the graph are likely to live nearby and have similar interests
- additionally, the interests of a user depend on her location; these interests also define the hashtags of posts that she creates
- the user's location determines with high probability her language, first and last names (common in that area), the university she attended and the company she works in (nearby institutions).

There is also a number of "trivial" (but not obvious to query optimizers) time constraints such as: users start posting messages in a forum only after they have joined it (which in turn only happens after that forum is created), replies to a message are created only after the message itself is published, etc. Moreover, the distributions of global properties of the dataset (e.g. frequency of names in any country, node degrees in the social graph) follow empirically observed laws (exponential distribution).
The number of such data correlations is fixed by the designers of the benchmark: the idea here is not to represent all of the real-world dependencies (that would be impossible), but to use this fixed set of correlating properties to construct challenging queries.

Methodology

LDBC benchmark development is driven by the notion of a choke point. A choke point is an aspect of query execution or optimization which is known to be problematic for the present generation of various DBMS (relational, graph and RDF). This concept is inspired by the classical TPC-H benchmark. Although TPC-H design was not based on explicitly formulated choke points, the technical challenges imposed by the benchmark's queries have guided research and development in the relational DBMS domain in the past two decades [16]. A detailed analysis of all choke points used to design the SNB Interactive workload is outside the scope of this brief description, the reader can find it in [81]. In general, the choke points cover the "usual" challenges of query processing (e.g., subquery unnesting, complex aggregate performance, detecting dependent group-by keys etc.), as well as some hard problems that are usually not part of synthetic benchmarks. Here we list a few examples of essential graph query processing challenges:

1) Estimating cardinality in graph traversals with data skew and correlations. As graph traversals are in fact repeated joins this comes back at a crucial open problem of query optimization in a slightly more severe form. SNB queries stress

cardinality estimation in transitive queries, such as traversals of hierarchies (e.g., made by replies to posts) and dense graphs (paths in the friendship graph). This problem is especially challenging given the correlations and skewed distributions in the SNB dataset, discussed above.

2) Choosing the right join order and type. This problem is directly related to the previous one, cardinality estimation. Moreover, as we have discussed in Chapter 4, there is an additional challenge for RDF systems where the plan search space grows much faster compared to equivalent SQL queries.

3) Handling scattered index access patterns. Graph traversals (such as neighborhood lookup) have random access without predictable locality, and efficiency of index lookup is very different depending on the locality of keys. Also, detecting absence of locality should turn off any locality dependent optimizations in query processing.

4) Parallelism and result reuse. All SNB queries offer opportunities for intra- and inter-query parallelism. Additionally, since most of the queries retrieve one- or two-hop neighborhoods of persons in the social graph, and the *Person* domain is relatively small, it might make sense to reuse results of such retrievals across multiple queries. This is an example of recycling: a system would not only cache final query results, but also intermediate query results of a "high value", where the value is defined as a combination of partial query result size, partial query evaluation cost, and observed frequency of the partial query in the workload.

Example. In order to illustrate the choke point-based design of SNB queries, we will describe technical challenges behind one of the queries in the Interactive workload, Query 9. Its definition in English is as follows:
Query 9: *Given a start Person, find the 20 most recent Posts/Comments created by that Person's friends or friends of friends. Only consider the Posts/Comments created before a given date.*
This query looks for paths of length two or three, starting from a given Person, moving to the friends and friends of friends, and ending at their created Posts/Comments. This *intended query plan*, which the query optimizer has to detect, is shown in Figure 6.2. Note that for simplicity we provide the plan and discussion assuming a relational system. While the specific query plan for systems supporting other data models will be slightly different (e.g., in SPARQL it would contain joins for multiple attributes lookup), the fundamental challenges are shared across all systems.
Although the join ordering in this case is fairly straightforward, an important task for the query optimizer here is to detect the types of joins, since they are

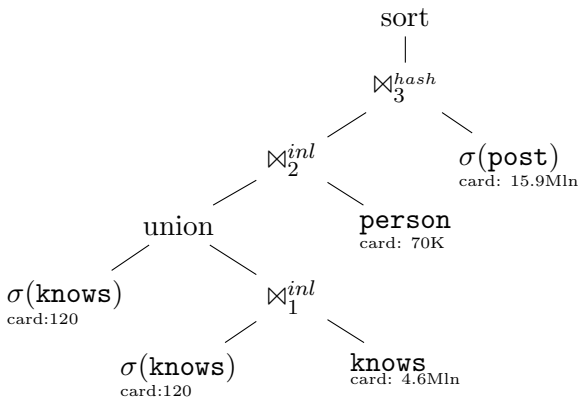


Figure 6.2.: Intended execution plan for Query 9

highly sensitive to cardinalities of their inputs. The lower most join $\bowtie_1$ takes only 120 tuples (friends of a given person) and joins them with the entire `Knows` table to find the second degree friends. This is best done by looking up these 120 tuples in the index on the primary key of `Knows`, i.e. by performing an *index nested loop join*. Same holds for the next $\bowtie_2$, since it looks up around a thousand tuples in an index on primary key of `Person`. However, the inputs of the last $\bowtie_3$ are too large, and the corresponding index is not available in `Post`, so Hash join is the optimal algorithm here. Note that picking a wrong join type hurts the performance here: in the HyPer database system, replacing index-nested loop with hash in $\bowtie_1$ results in 50% penalty, and similar effects are observed in the Virtuoso RDBMS.
Determining the join type in Query 9 is of course a consequence of accurate *cardinality estimation* in a graph, i.e. in a dataset with power-law distribution. In this query, the optimizer needs to estimate the size of second-degree friendship circle in a dense social graph.
Finally, this query opens another opportunity for databases where each stored entity has a unique synthetic identifier, e.g. in RDF or various graph models. There, the system may choose to assign identifiers to `Posts/Comments` entities such that their IDs are increasing in time (creation time of the post). Then, the final selection of Posts/Comments created before a certain date will have high locality. Moreover, it will eliminate the need for sorting at the end.

Interactive Workload

Three workloads are supposed to constitute the Social Network Benchmark: *Interactive Workload*, *Business Intelligence*, *Graph Analytics*, although at the time of writing, only the first workload is finalized, the two remaining are under

development. All three of them will be based on the same dataset and use the common benchmark infrastructure such as the benchmark driver and validation tools. We will describe the Interactive workload here.
The Interactive Workload mimics a user's behavior on the online social network site. Users browse the content, add friends, write posts or comments and occasionally ask user-centric read queries, such as *What are the new trends that my friends have been posting about in the last days?*. From the database perspective, the Interactive Workload tests ability of the System Under Test (SUT) to cope with a transactional workload, composed from short updates, short lookups and relatively long read queries. The latter type of queries distinguishes this benchmark from purely transactional benchmarks like TPC-C. An important characteristic of all queries and updates is that they touch only limited parts of the dataset: at the very maximum it is information about friends of friends of a given user (required by some of the user-centric read queries), and typically it is just properties of a given user (touched by updates and short reads).
The core query optimization challenges in the workload are represented by 14 read queries, shown in the Appendix. They retrieve information about the social environment of a given user (one- or two-hop friendship circle), such as new groups that the friends have joined, new hashtags that the environment has used in recent posts etc. Although they represent plausible questions that a user of a real social network may need, their complexity is typically beyond the functionality of modern social network providers due to their online nature (e.g., we do not assume any pre-computation). These queries present the core of query optimization choke points in the benchmark. LDBC website[1] and [81] have formulations of these queries in SQL, SPARQL, Cypher and APIs of two graph databases (Neo4j and Sparksee).

6.2. Motivation for Parameter Curation

A typical benchmark consists of two parts: (i) the dataset, which can be either real-world or synthetic, and (ii) the workload generator that issues queries against the dataset based on the pre-defined *query templates*. A query template is an expression in the query language (e.g., SQL or SPARQL) with *substitution parameters* that have to be replaced with real bindings by the workload generator. For example, a template of LDBC-Interactive Query 5 that finds new groups that friends and friends of friends of a given user have joined recently, looks like this:
In a query workload, the workload driver would execute this query template in

[1] `ldbcouncil.org/developer/snb`

```
select top 20 f_title, f_forumid, count(*)
from forum, post, forum_person,
 ( select k_person2id
   from knows
   where k_person1id = %Person%
   union
   select k2.k_person2id
   from knows k1, knows k2
   where k1.k_person1id = %Person%
     and k1.k_person2id = k2.k_person1id
     and k2.k_person2id <> %Person%
 ) f
where f_forumid = ps_forumid and f_forumid = fp_forumid
   and fp_personid = f.k_person2id
   and ps_creatorid = f.k_person2id
   and fp_creationdate >= '%Date0%'::date
group by f_forumid, f_title
order by 3 desc, f_title
```

Query 6.1: LDBC-Interactive Query 5

one experiment potentially multiple times (e.g., 100) with different bindings for the *%Person* and *%Date0* parameters. It would report an aggregate value of the observed runtime distribution per query (usually, the average runtime per query template). This aggregated score serves two audiences: First, the users can evaluate how fit a specific system is for their use-case (choosing, for example, between systems that are good in complex analytical processing and those that have the highest throughput for lookup queries). Second, database architects can use the score to analyze their systems' handling of choke points, discussed in Section 6.1, like handling multiple interesting orders or sparse foreign key joins. In "throughput" experiments, the benchmark driver may also execute the above experiment multiple times in multiple concurrent query streams. For each stream, a different set of parameters is needed.

Desired Properties. In order for the aggregate runtime to be a useful measurement of the system's performance, the selection of parameters for a query template should guarantee the following properties of the resulting queries:

P1: the query runtime has a bounded variance: the average runtime should correspond to the behavior of the majority of the queries

P2: the runtime distribution is stable: different samples of (e.g., 10) parameter bindings used in different query streams should result in an identical runtime distribution across streams

P3: the optimal logical plan (optimal operator order) of the queries is the same: this ensures that a specific query template tests the system's behavior under the well-chosen technical difficulty (e.g., handling voluminous joins or proper cardinality estimation for subqueries etc.)

The conventional way to get the parameter bindings for *%Person* is to sample the values (uniformly, at random) from all the possible person IDs in the dataset (the "domain"). This is, for example, how the TPC-H benchmark creates its workload. Since the TPC-H data is generated with simple uniform distribution of values, the uniform sample of parameters trivially guarantees the properties **P1-P3**. The TPC-DS benchmark moved away from uniform distributions and uses "step-shaped" frequency distributions instead [79, 94], where there are large differences in frequency between steps, but each step in the frequency distribution contains multiple values all having the same frequency. This allows TPC-DS to obtain parameter values with exactly the same frequency, by choosing them all from the same step.

However, these techniques do not work for LDBC SNB, since it uses a dataset with skewed value distribution and close-to-realistic correlations between values. Also, uniform sampling does not perform well for benchmarks that use real-world datasets (we provide an example against IMDB schema in Section 6.2.1), since by their nature they frequently feature non-uniform distributions of values. In our example above, the behavior of the query changes significantly depending on the selection of the parameter. We present a detailed picture of its runtime behavior in Section 6.2.1, but most notably, if *%Person* is a highly connected user (with hundreds of friends), the query features a voluminous join between friends of friends and `Forum`, while for "less popular" users the join is rather sparse. As we see, two very different scenarios are tested for these two parameter choices, and they should ideally be reported separately. The *%Person* parameter bindings for these two scenarios would be drawn from two buckets of persons, with large number of friends and with a few friends.

We clarify that our intention is not to obviate the interesting query optimization problems related to the real-world distributions and correlations in the dataset, but to make the results within one query template predictable by choosing the parameters that satisfy properties **P1**-**P3**, in order to guarantee that the behavior of the System Under Test (SUT) and of the benchmark results is *understandable*. In case different parameters have very different runtimes and optimal query plans (e.g. due to skew or correlations) this can still be tested in a benchmark by having multiple query *variants*, e.g., one variant with persons with many friends, another with users that have few friends. The different variants would behave very differently and test whether the optimizer makes

good decisions, but within the same query variant the behavior should be stable and understandable regardless the substitution parameter.

Parameter Curation. In this section we present an approach to generate parameters that yield similar behavior of the query template, which we coin "Parameter Curation". We consider a setup with a fixed set of query templates and a dataset (either real-world or synthetic) as input for the parameter generator. Our approach consists of two parts:

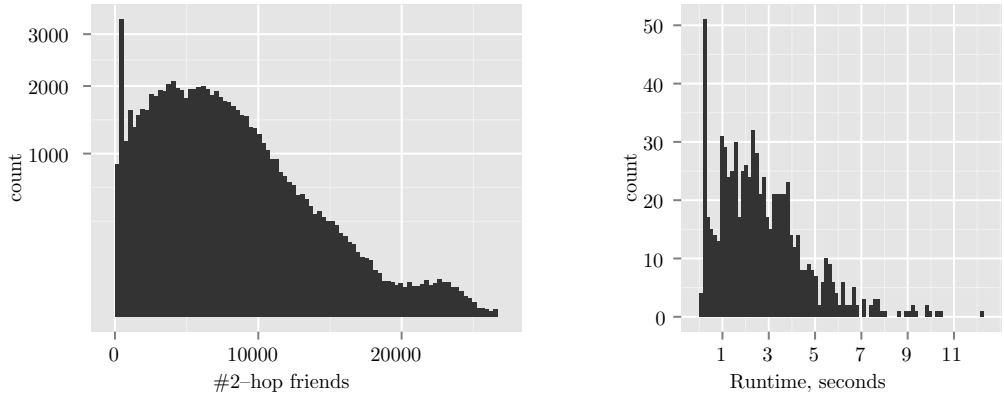
- for each query template for all possible parameter bindings, we determine the size of intermediate results in the *intended* query plan. Intermediate result size heavily influences the runtime of a query, so two queries with the same operator tree and similar intermediate result sizes at every level of this operator tree are expected to have similar runtimes. This analysis on result sizes versus parameter values is done once for every query template (remember that we consider benchmarks with a *fixed* set of queries).
- we define a greedy algorithm that selects ("curates") those parameters with similar intermediate result counts from the dataset.

Note that Parameter Curation depends on data generation in a benchmark: we are mining the generated data for suitable parameters to use in the workload. As such, Parameter Curation constitutes an new phase that follows data generation in a typical database benchmarking process.

6.2.1. Examples

For all examples in this section, we use the SNB-Interactive workload, where we generated a social network of scale factor 30 (ca. 30 GB of CSV files). For both datasets we use Virtuoso 7 database (Column store) and run our experiments on a commodity server with the following specifications: Dual Intel X5570 Quad-Core-CPU, 64 Gb RAM, 1 TB SAS-HD, Redhat Enterprise Linux (2.5.37).
In the following examples (**E1-E4**), we illustrate our statement that uniform selection of parameters leads to unpredictable behavior of queries, which makes interpretation of benchmark results difficult.

E1: Runtime distribution has high variance. When drawing parameters uniformly at random, we encounter a very skewed runtime distribution for queries over real-world datasets. We plot the runtime of Query 6.1 (SNB-Interactive Query 5) for a uniform sample of 100 *%Person* parameters in Figure 6.3b. As we see, the difference between the fastest and slowest query is more than 100



(a) Distribution of size of 2-hop environment in SNB graph

(b) Query 5 runtime distribution

Figure 6.3.: Correlations cause high runtime variance (LDBC Query 5)

times. The reason for this is the distribution of the size of two-hop friendship environment in the SNB graph: since the number of friends has a power-law distribution, the number of friends of friends follows a multimodal distribution with several peaks, see Figure 6.3a. This translates into highly variable amount of data that the query needs to touch depending on the parameter *%Person*, which in turn influences the runtime.

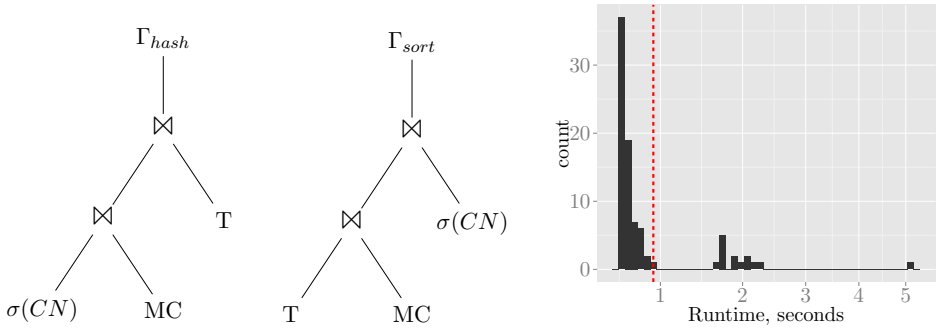
E2: Different plans for different parameters. The uniformly generated parameter bindings can lead to completely different plans for the same query template. This happens because the cardinalities of the subqueries naturally depend on the parameter bindings, and sometimes on the combination of the parameters. To illustrate this case, we consider the following query for IMDB dataset, finding all the movie producing companies from the country *%Country* that have released more that 20 movies:

Query 6.2: IMDB Query

```
select cn.name, count(t.id) cnt
from title t, movie_companies mc, company_name cn
where t.id = mc.movie_id and cn.id = mc.company_id
  and cn.country_code = '%Country%' and t.kind_id = 1
group by mc.company_id, cn.name
having count(*) > 20
order by cnt desc
limit 20
```

Two optimal plans for Query 6.2 (as found by the PostgreSQL database) are

depicted in Figure 6.4a) and b), where leaves are marked with table aliases from the query listing. Picking 'US' – the country with the largest number of companies – as a parameter not only changes the join order, as compared with the 'UK' parameter, but also results in applying a different group-by method (by sorting as opposed to hash-based grouping for the 'UK' parameter).



(a) %Country = 'UK' **(b)** %Country = 'US' **(c)** Runtime distribution, red line marks the mean

Figure 6.4.: IMDb Query 6.2 plans and runtime distribution for different parameters

As another example, we consider LDBC Query 3 that *finds the friends and friends of friends that have been to countries X and Y*. The optimal plan for this query can start either with finding all the friends within two steps from the given person, or from extracting all the people that have been to countries X and Y: if X and Y are Finland and Zimbabwe, there are supposedly very few people that have been to both, but if X and Y are USA and Canada, this intersection is very large. In the SNB benchmark, correlations that might not even be detected by the optimizer aggravate the execution picture beyond plain frequency differences. As discussed in Section 6.1, there is a correlation between the location of each user and her friends (they often live in the same country) and travel destinations are correlated so that nearby travel is more frequent. Hence combinations of countries far from home are extremely rare and combinations of neighboring countries frequent.

We note that the plan variability is not a bad property *per se*: indeed, this query forces the query optimizer to accurately estimate the cardinalities of subqueries depending on input parameters. However, the generated parameters should be sampled independently for two different variants (countries that are rarely and frequently visited together), to allow a fair and complete comparison of different query optimization strategies.

E3: Average runtime is not representative. In addition to being far from

uniform (**E1**), the query runtime distribution can also be "clustered": depending on the parameter binding, the query runs either extremely fast or surprisingly slow, and the average across the runtimes does not correspond to any actual query performance. To illustrate this issue, we consider again the IMDB Query 6.2. Figure 6.4c shows the runtime distribution of that query over the entire domain of *%Country* parameter bindings. We see that the average runtime (red line on the plot) falls outside of the larger group of parameter bindings, so in fact very few actual queries have the runtime close to the mean.

E4: Sampling is not stable. A single query in the benchmark is typically being executed several times with different randomly chosen parameter bindings. It is therefore interesting to see how the reported average time changes when we draw a different sample of parameters. In order to study this, we take Query 2 of the LDBC benchmark that *finds the newest 20 posts of the given user's friends*. We sample 4 independent groups of parameter bindings (100 user parameter bindings in each group), run the query with these parameters and report the aggregated runtime numbers within individual groups (q_{10} and q_{90} are the 10th and the 90th percentiles, respectively).

Time	Group 1	Group 2	Group 3	Group 4
q_{10}	0.14 s	0.07 s	0.08 s	0.09 s
Median	1.33 s	0.75 s	0.78 s	1.04 s
q_{90}	4.18 s	3.41 s	3.63 s	3.07 s
Average	1.80 s	1.33 s	1.53 s	1.30 s

We see that uniform at random generation of query parameters in fact produces unstable results: if we were to run 4 workloads of the same query with 100 different parameters in each workload, the deviation in reported average runtime would be up to 40%, with even stronger deviation on the level of percentiles and median runtime (up to 100%). When TPC-H benchmark record results are improved, this often only concerns minor difference with the previous best (e.g. 5%). Hence, the desired stability between different parameter runs of a benchmark should ideally have a variance below that ballpark.

6.3. Problem Definition

Here we define the problem of generating the parameter binding for benchmark queries. In order to compare two query plans formulated in logical relational algebra, we use the classical logical cost function that takes into account the sum of intermediate results produced during the plan's execution [73]:

$$C_{\text{out}}(T) = \begin{cases} |R_x| & \text{if } T \text{ is a scan of relation } R_x \\ |T| + C_{\text{out}}(T_1) + C_{\text{out}}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

The above formula is incomplete and just here for argumentation; a more complete version of this logical cost formula naturally should include all relational operators (hence also selection, grouping, sorting, etc). The main idea is that for every relational operator T_y it holds the amount of tuples that pass through it. In our experiments, the cost function C_{out}, which is computed using the de-facto result sizes (not the estimates!), strongly correlates with query running time (ca. 85% Pearson correlation coefficient). Therefore, if two query plan instances have the same C_{out}, or even better if all operators in the query plan have the same C_{out}, these plans are expected to have very similar running time.
In order to find k parameter bindings that yield identical runtime behavior of the queries, we could:

a: enumerate the set of all equivalent logical query plans L_Q for a query template Q.

b: for each possible parameter p from domain P, and each subplan T_{lq} of L_Q compute $C_{out}(T_{lq}(p))$.

c: find subset $S \subset P$, with size $|S| = k$, such that the sum of all variances $\sum_{\forall T_{lq} \in L_Q} \mathit{Variance}_{\forall p \in S}\ C_{out}(T_{lq}(p))$ is minimized.

Note that this generic problem of parameter curation is infeasibly hard to solve. The amount of possible query plans is exponential in the amount of operators (e.g. $2^{|L_Q|}$, just for leftdeep-only plans, and $|L_Q|$ being the amount of operators in plan L_Q), and all these plan costs would have to be calculated very many times: for each possible set of parameter bindings (whose size is $2^{|P|}$, where $|P|$ is the product of all parameter domain sizes – a typically quite large number), and for all $|L_Q|$ subplans of L_Q.
Instead, we simplify the problem by focusing on a single *intended* logical query plan. Since we are designing a benchmark, which consists of a relatively small set of query templates (the intended benchmark workload), and in this benchmark design we have certain intentions, this is feasible to do manually. We can, therefore, formulate a more practical problem of Parameter Curation as follows:

Parameter Curation: For the Intended Query Plan QI and the parameter domain P, select a subset $S \subset P$ of size k such that $\sum_{\forall T_{qi} \in QI} \mathit{Variance}_{\forall p \in S}\ C_{out}(T_{qi}(p))$ is minimized.

Since the cost function correlates with runtime, queries with identical optimal

plans w.r.t. C_{out} and similar values of the cost function are likely to have close-to-normal distribution of runtimes with small variance. Therefore, the properties **P1-P3** from Section 6.2 hold within the set of parameters S and effects mentioned in Section 6.2.1 are eliminated.
The Parameter Curation problem is still not trivial. A possible approach would be to use query cardinality estimates that an EXPLAIN feature provides. For each query template Q we could fix the operator order to the intended order QI, run the query optimizer for every parameter p and find out the estimated $C_{out}(QI(p))$, and then group together parameters with similar values. However, it seems unsatisfactory for this problem, since even the state-of-the-art query optimizers are often very wrong in their cardinality estimates. As opposed to estimates we will therefore use the de-facto amounts of intermediate result cardinalities (which are otherwise only known after the query is executed).

6.4. Algorithms for Parameter Curation

In this section we demonstrate how the problem of Parameter Curation for a given query plan is solved in several important cases, namely:

- a query with a single parameter
- a query with two (potentially correlated) parameters, one from discrete and another from continuous domain. Such a combination of parameters could be: *Person* and *Timestamp* (of her posts, orders, etc).
- multiple (potentially correlated) parameters, such as *Person*, her *Name* and the *Country* of residence.

Note that our solution easily generalizes to the cases of multiple parameters (such as two *Timestamp* parameters etc); we consider the simplest cases merely for the purposes of presentation.
Our solution is divided into two stages. First, we perform *data analysis* that aims at computing the amount of intermediate results produced by the given query execution plan across the entire domain of parameter(s). The output of the analysis is a set of parameter(s) values and the corresponding intermediate result sizes produced by every join of the query plan. Second, the output of the data analysis stage is processed by the *greedy algorithm* that selects the subset of parameters resulting in the minimal variance across all intermediate result sizes.

6.4.1. Single Parameter

Data Analysis The goal of this stage is to compute all the intermediate results in the query plan for each value of the parameter. We will store this information as a *Parameter-Count* (PC) table, where rows correspond to parameter values, and columns – to a specific join's result sizes.

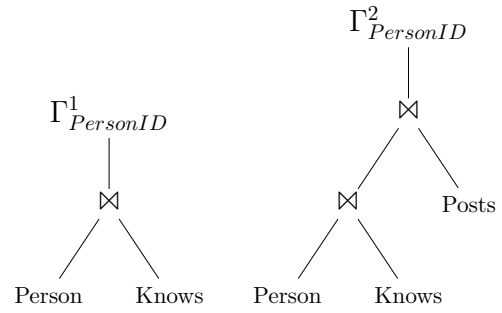
There are two ways of computing that table. First, given the query plan tree we can split it into a bottom-up manner starting with the smallest subtree that contains the parameter. We will then remove the selection on the parameter value from the query, and add a Group-By on the parameter name with a Count, thus effectively aggregating the result size of that subtree across the parameter domain. In our experiments with SNB-Interactive we were generating group-by queries based on the JSON representation of the query plan.

The second way of computing the Parameter-Count table is to compute the corresponding counts as part of data generation. Indeed, in case of the SNB benchmark, for instance, all the group-by queries boil down to counting the number of generated entities: number of friends per person, number of posts per user etc. These counts are later used to generate parameters across multiple queries.

As an example, consider a simplified version of LDBC Query 2, given in Listing 6.3, which extracts 20 posts of the given user's friends ordered by their timestamps. The generated plans with Group-By's on top are depicted in Figure 6.5a and b. The first subquery plan counts the number of friends per person, the second one aggregates the number of posts of all friends by user. The resulting Parameter-Count table is given in Figure 6.5c, where columns named $|\Gamma^1|$ and $|\Gamma^2|$ correspond to the results of the first and second group-by queries, respectively. In other words, when executed with $\%ParameterID = 1542$, Query 2 will generate $60 + 99 = 159$ intermediate result tuples.

```
select p_personid, ps_postid, ps_creationdate
from person, post, knows
where
    person.p_personid = post.ps_creatorid and
    knows.k_person1id = %Person% and
    knows.k_person2id = person.p_personid
order by ps_creationdate desc
limit 20
```

Query 6.3: LDBC Query 2



(a) Step 1: # Friends per Person

(b) Step 2: # Posts of Friends

PersonID	$\lvert\Gamma^1\rvert$	$\lvert\Gamma^2\rvert$
...	...	...
1542	60	99
1673	60	102
7511	60	103
958	61	120
1367	61	101
...	...	...

(c) Parameter-Count table

Figure 6.5.: Preprocessing for the query plan with a single parameter

Greedy Algorithm. Now, our goal is to find the part of the Parameter-Count table with the smallest variance across all columns. Note that the order of the columns matters; in other words, variance in the first column (result size of the bottom-most join of the query plan) is more crucial to the runtime behavior than variance in the last column (top-most join). Following this observation, we construct a simple greedy algorithm, depicted in Algorithm 18. It uses an auxiliary function `FindWindows` that finds the *windows* (consecutive rows of the table) of size at least k on a given column i with the smallest possible variance (lines 3-4). In our table in Figure 6.5c such windows on the first column ($|\Gamma^1|$) are highlighted with red and green colors (they consist of parameter sets $[1542, 1673, 7511]$ and $[958, 1367]$, respectively). Both these sets have variance 0 in the column $|\Gamma^1|$.

The algorithm starts with finding the windows $\mathbb{W}$ with the smallest variance on the entire first column (line 9). Then, in every found window from $\mathbb{W}$ we look for smaller sub-windows (but of size at least than k, see line 3) that minimize variance on the second column (lines 12-16). The found windows with the smallest variance become candidates for the next iteration, based on further columns (line 17). The process stops when we reach the last column or the number of candidate windows reduces to 1.

In the example from Figure 6.5c, the first iteration brings the two windows mentioned above (red and green). Then, in every window we look for windows of k rows, they are $[99, 102]$, $[102, 103]$ and $[120, 101]$. Out of these three candidates, $[102, 103]$ has the smallest variance (highlighted in blue), so our solution consists of two parameters, $[1673, 7511]$.

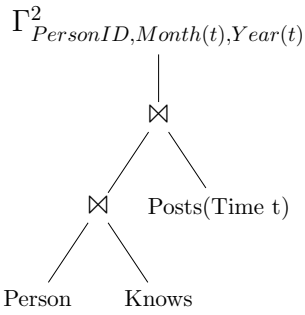
Algorithm 14: Parameter Curation (single parameter)

```
FindWindows
Input: PC – Parameter-Count table, i – column, start, end – offsets in the
       table
1  begin
2      scan the PC table on the ith column from start to end rows
3      W ← generate Windows of size K
4      merge overlapping windows with the same variance
5      return w ∈ W with the smallest variance of PC[i] values
6  ParameterCuration
   Input: PC – Parameter-Count table, n – number of count columns in PC
   Result: 𝕎 – window in PC table with the smallest variance of counts
           across all columns
7  begin
8      i ← 1 ▷ corresponds to the column number in the table, i.e. |Γ^i|
9      𝕎 ← FindWindows(PC, 1, 0, |PC|) ▷ find windows on the entire first column
10     while |𝕎 > 1| and i < n do
11         i ← i + 1
12         𝕎_new ← list()
13         for w ∈ 𝕎 do
14             w' ← FindWindows(PC, i, w.start, w.end)
15             𝕎_new.add(w')
16         sort 𝕎_new by variance asc
17         𝕎 ← all w ∈ 𝕎_new with the smallest variance
18     return 𝕎
```

6.4.2. Two Correlated Parameters

Here we consider the case when a query has two parameters, discrete and continuous, e.g. *PersonID* and *Timestamp*. The continuous parameter is involved in a selection, e.g. specifying the time interval. We focus on the situation when these two are correlated, otherwise the solution of the Parameter Curation problem is a straightforward generalization of the previous case: one would follow the independence assumption and find the bindings for the discrete parameter using Parameter-Count table, and then select intervals of the same length as bindings of the continuous parameter.

However, if parameters are correlated, the independence assumption may lead to a significant skew in the C_{out} function values. We take the LDBC Query 2 as an example again, which in its full form also includes the selection on the timestamp of the posts `ps_creationdate < %Date0%` (i.e., the query *finds the top 20 posts*



(a) # Posts of Friends By Month

PersonID	Jan'14	Feb'14	Mar'14	Apr'14
...	...	...	...	...
1673	0	30	30	42
7511	20	30	30	23
...	...	...	...	...

(b) Parameter-Count table with Time buckets ($PCTime$)

Figure 6.6.: Preprocessing for the query plan with two correlated parameters

of friends of a user written before a certain date). In the LDBC dataset, the *PersonID* and *Timestamp* of the user's posts are naturally correlated, since users join the modeled social network at different times; moreover, their posting activity changes over time. Therefore, if we choose the Timestamp parameter in LDBC Query 2 independently from the *PersonID*, the amount of intermediate results may vary significantly (even if ParameterIDs were curated such that the total number of posts is the same).

Data analysis. In order to capture the correlation between two parameters, we need to include the second one (Timestamp in our example) in the grouping key during the Parameter-Count table construction. Grouping by the continuous parameter may lead to a very large and sparse table, so we "bucketize" it (e.g., by months and years for Timestamp). We then store the results of the aggregation as a Parameter-Count table, along with the bucket boundaries.
Our example from Figure 6.5 is extended with the Timestamp parameter in Figure 6.6. The partial join trees are complemented with additional Group-By on Month and Year of the timestamp as soon as the corresponding table containing the Timestamp (in our case *Posts*) is added to the plan (in this example, at Step 2 when we consider the second join). Assuming that our dataset spans 4 months of 2014, the resulting table may look like Figure 6.6b.

Greedy algorithm. The first stage of the Parameter Curation for two parameters ignores the continuous parameter (e.g. Timestamp). As a result, we get the bindings for the first (discrete) parameter that have similar intermediate result sizes across the entire domain of the continuous parameter. Now for these curated parameter bindings we find the corresponding continuous parameters such that the C_{out} function values are similar across all the curated parameters.

For the purpose of presentation we consider the solution for the *%Date0* parameter that appears in the selection of a form $timestamp < \%Date0$. In our example from the previous section, we have found two *PersonID* parameters that have the smallest variance in C_{out}. Let $PCTime[i, j]$ denote the count in the Parameter-Count table for the parameter i in bucket j, and N be the number of buckets for continuous parameter. For example, in Figure 6.6b $PCTime[1673, Mar'14] = 30$ is the number of posts made by friends of the user 1673 in March 2014, and $N = 4$.

- We compute the partial sums of the monthly counts $Sum[i] = \sum_{j=1..N-M} PCTime[i, j]$ for all the discrete parameter bindings i for all the months except the last M (where M is typically 1..3). In the table in Figure 6.6b for $M = 1$ these partial sums are 60 and 80 for *PersonID*s 1673 and 7511, respectively.

- We determine the average $\mathcal{A}$ across these sums $Sum[i]$ (70 in our example)

- For every discrete parameter i we pick the bucket J such that $\sum_{j=1..J} PCTime[i, j]$ is as close as possible to the global average $\mathcal{A}$. More precisely, we pick the first bucket such that the sum exceeds the global average. In our example, for $i = 1673$, J is the fourth bucket (*Apr'14*)

- Finally, since our buckets represent continuous variable (time), we can split the bucket J so that the sum of counts is *exactly* $\mathcal{A}$. For $i = 1673$ we need to get 10 posts in April 2014 (60 are covered by previous months, and we need to reach the global average of 70). We pick April $\frac{42 \cdot 10}{30} = 14$ as $Date0$.

In order to perform the last step in the above computation, we have assumed that within one bucket the count is uniformly distributed (e.g., every day within one month has the same number of posts). Even when this assumption does not hold precisely, the effects are usually negligible.
The timestamp conditions of a different form, e.g. $Timestamp > Date0$, or $Timestamp \in [Date0, Date1]$ are handled in the same manner. For example, the $Timestamp \in [Date0, Date1]$ condition leads to finding for every $PersonID$ the median of its post-per-time distribution, that is the median of the $PCTable[i, j]$ for every row i. Then, the median of those medians is identified across all *PersonID*s, and finally every individual *PersonID*'s median is made as close as possible to the global median by extending/reducing the corresponding bucket.

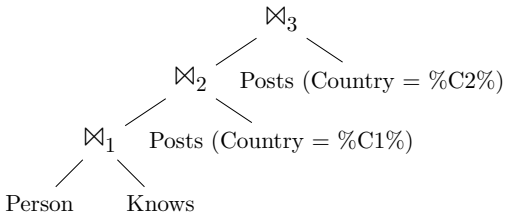
6.4.3. Multiple Correlated Parameters

Parameter Curation for multiple (more than two) parameters follows the scheme of two parameters: one is selected as a primary (*PersonID*), the other ones are "bucketized". This way we get sets of bindings, each of those results in identical query plan and similar runtime behavior.
In case of correlated parameters, however, it may be interesting to find several sets of parameter bindings that would yield different query plans (but consistent within one set of bindings). Consider the simplified version of LDBC Query 3 that is *finding the friends of a user that have been to countries %C1 and %C2 and logged in from that countries (i.e., made posts)*, given in Query 6.4 and its query plan in Figure 6.7a.

```
select k.k_person2id, ps_postid, ps_creationdate
from person p, knows k, post p1, post p2
where p.person_id = k.k_person1id
                and k.k_person2id = p1.p_personid
                and k.k_person2id = p2.p_personid
                and p1.place = '%C1%'
                and p2.place = '%C2%'
order by ps_creationdate desc
limit 20
```

Query 6.4: LDBC Query 3

Since in the generated LDBC dataset the country of the person is correlated with the country of his friends, and users tend to travel to (i.e. post from) neighboring countries, there are essentially two groups of countries for every user: first, the country of his residence and neighboring countries; second, any other country. For parameters from first group the join denoted $\bowtie_2$ in Figure 6.7a becomes very unselective, since almost all friends of the user are likely to post from that the country. For the second group, both $\bowtie_2$ and $\bowtie_3$ are very selective. In the intermediate case when parameters are taken from the two different groups, it additionally influences the order of $\bowtie_2$ and $\bowtie_3$.
Both these groups of parameters are based on counts of posts made by friends of a user, i.e. based on the counts collected in the Parameter-Count table (with additional group-by on country of the post). Instead of keeping the buckets of all countries, we group them into two larger buckets based on their count, *Frequent* and *Non-Frequent* as shown in Figure 6.7b.
Now we can essentially split the LDBC Query 3 into three different (related) query variants (a), b) and c)), based on the combination of the two *%Country* parameters: a) *%C1* and *%C2* from the *Frequent* group, b) both from *Non-*



(a) Query Plan

Countries

PersonID	Freq	Non-Freq
...	...	...
1673	CH, JP, IN	ES, FR, SE
7511	DE, FR, IT	CA, BR, AU
...	...	...

(b) Parameter-Count table

Figure 6.7.: Case of multiple correlated parameters

Frequent group, c) combination of the two above.

6.5. Experiments

In this section we describe our experiments with curated parameters in the SNB-Interactive workload. First, we compare the runtimes of query templates with curated parameters as opposed to randomly selected ones. Then we proceed with an experiment on curating parameters for different intended plans of the same query template in Section. All experiments are run with Virtuoso 7 Column Store as a relational engine on a commodity server.

Curated vs Uniformly Sampled Parameters

First experiment aims at comparing the runtime variance of the LDBC queries with curated parameters with the randomly sampled parameters. For all 14 queries we curated 500 parameters and sampled randomly the same amount of parameters for every query. We run every query template with each parameter binding for 10 times and record the mean runtime. Then, the compute the runtime variance per query for curated and random parameters. The results, given in Table 6.1, indicate that Parameter Curation reduces the variance of runtime by a factor of at least 10 (and up to several orders of magnitude). We note that some queries are more prone to runtime variability (such as Query 4 and 5), that is why the variance reduction is different accross the query set. For Query 4 we additionally report the runtime distribution of query runs with curated and random parameters in Figure 6.8.

Groups of Parameters for One Query

So far we have considered the scenario when the *intended query plan* needs to be supplied with parameters that provide the smallest variance to its runtime. For some queries, however, there could be multiple intended plan *variants*, especially

Query	Curated	Random
1	13	773
2	31	2165
3	243	444174
4	0.6	$184 \cdot 10^6$
5	1300	$52 \cdot 10^6$
6	6931	278173
7	33	362
8	0.18	403
9	99269	880287
10	4073	102852
11	1	39
12	95	1535
13	2977	26777
14	5107	155032

Table 6.1.: Variance of runtimes: Uniformly sampled parameters vs Curated parameters for the LDBC Benchmark queries

when the query contains a group of correlated parameters. As an example, take LDBC Query 11 that *finds all the friends of friends of a given person P that work in country X*. The data generator guarantees that the location of friends is correlated with the location of a user. Naturally, when the country X is the user's country of residence, the amount of intermediate results is much higher than for any other country. Moreover, if X is a non-populous country, the reasonable plan would be to start from finding all the people that work at organizations in X and then figure out which of them are friends of friends of the user P.

As described in Section 6.4.3, our algorithm provides three sets of parameters for the three intended query plans that arise in the following situations: (i) P resides in the country X, (ii) country X is different than the residence country of P, (iii) X is a non-populous country that is not a residence country for P. As a specific example, we consider a set of Chinese users with countries (i) China, (ii) Canada, (iii) Zimbabwe. The corresponding average runtimes and standard deviations are depicted in Figure 6.9. We see that the three groups indeed have distinct runtime behavior, and the runtime within the group is very similar. For comparison, we also provide the runtime distribution for a randomly chosen country parameter, which is far from the normal distribution.

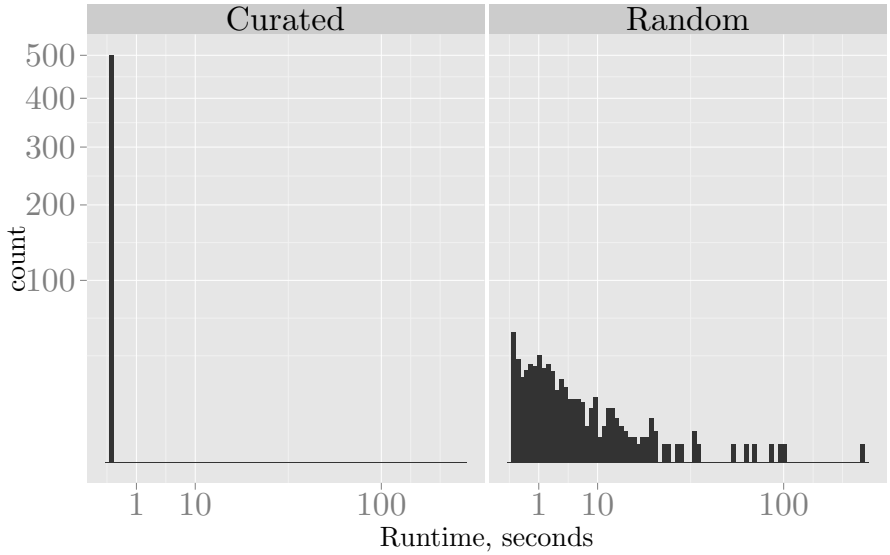


Figure 6.8.: LDBC Query 4 runtime distribution: curated vs random parameters

	10K	50K	1M
Parameter Extraction Time	17 s	125 s	4329 s
% of Total Generation Time	7%	11%	12%
Data size, Gb	1	5.5	227

Table 6.2.: Time to extract parameters in the LDBC datasets of different scales

Parameter Curation time

Finally, we report the runtime of the parameter curation procedure for the LDBC Benchmark. Note that we have incorporated the data analysis stage in our case is implemented as part of data generation, e.g. we keep the number of posts per person generated, number of replies to the user's posts etc. This is done with a negligible runtime overhead. In Table 6.2 we report the runtime of the greedy parameter extraction procedure for the LDBC dataset of different scales (as number of persons in the generated social network). We additionally show the size of the generated data; this is essentially an indicator of the amount of data that the extraction procedure needs to deal with. We see that Parameter Curation takes approximately 7% to 12% of the total data generation time, which looks like a reasonable overhead.

Our results show that Parameter Curation in the skewed and correlated datasets transforms chaotic performance behavior for the same query template with randomly chosen substitution parameters into highly stable behavior for curated

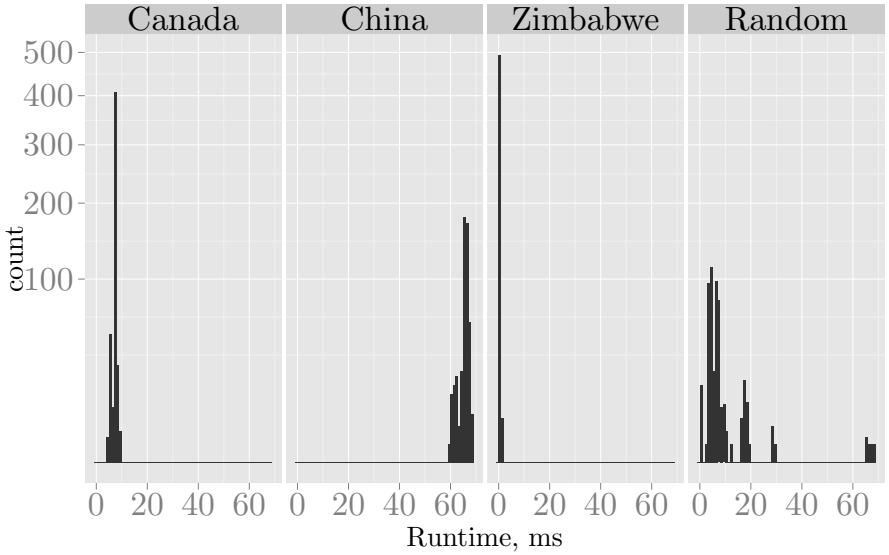


Figure 6.9.: LDBC Query 11 with four different groups of parameters (for countries China, Canada, Zimbabwe, Random)

parameters. Parameter Curation achieves its purpose: all the instances of queries from the same template get approximately equal runtimes (and therefore equal CPU share and contribution to the benchmark score). It retains the possibility for benchmark designers to test the ability of query optimizers to identify different query plans in case of skew and correlation, by grouping parameters with the same behavior into a limited number of classes which among them have very different behavior; hence creating multiple *variants* of the same query template. Our approach to focus the problem on a single *intended* query plan for each template variant reduces the high complexity of generic parameter curation. We experimentally showed that group-by based *data analysis* followed by *greedy parameter extraction* that implements Parameter Curation in the case of LDBC SNB is practically computable and can form the final part of the database generator process.

Conclusions

In this thesis we have covered multiple graph-related problems, from efficiently discovering shortest paths to optimizing declarative graph pattern matching queries. The major contributions of this work include novel techniques for query processing and query optimization in RDF databases (shortest path queries, complex graph pattern matching queries), as well as new indexes for approximate shortest path for the small-world graphs.

Apart from RDF data model, we have considered query optimization for the Property Graph data model. Its growing popularity is caused by its intuitive yet expressible query language and data format. In order to fully exploit the expressive power of a graph query language in an efficient manner, however, a major query optimization effort is required. Our work on a basic query optimizer for Cypher, described in Chapter 5, can serve as a foundation for full-fledged query optimizers for property graphs.

Future work should cover several directions: First, the cardinality estimation techniques can be significantly improved. In general, we only capture some significant structural phenomena in the graph (cf. indexing reachable nodes and paths in Chapter 2, star-shaped and chain-shaped subgraphs in Chapter 5). More subtle, yet sometimes crucial, correlations occur between the graph structure and the attributes of the nodes. We believe that the SNB benchmark described in Chapter 6 will motivate system architects to pay more attentions to these cases. Second, some applications for graph algorithms (in particular social network analytics) should be considered in more detail, since they may greatly benefit from specialized graph indexes and algorithms. Finally, some use cases do require distributed graph databases, and we believe that it is possible to extend our query optimization techniques (detecting structural correlations in graphs, query simplification, path indexing) into a distributed setup.

APPENDIX A

Path queries

Queries for the YAGO2 dataset

All queries use the following prefix:
yago: <http://yago-knowledge.org/resource/>

Query 1

```
 select ?loc ??path
 where {yago:Ulm ??path ?loc.
    pathfilter(containsOnly(??path, yago:isLocatedIn))
}
```

Query 2

```
select ?obj  ??loc
where {?obj ??loc yago:Germany.
   ?obj rdf:type yago:wordnet_mountain_109359803.
   pathfilter(containsOnly(??loc,yago:isLocatedIn))
}
```

Query 3

```
select ?person
where {?place ??loc yago:Germany.
   ?person yago:wasBornIn ?place.
   ?person yago:diedIn ?place.
   pathfilter(containsOnly(??loc,yago:isLocatedIn))
}
```

Query 4

```
select ?person
where {?place1 ??loc1 yago:Germany.
   ?place2 ??loc2 yago:France.
   ?person yago:wasBornIn ?place1.
   ?person yago:diedIn ?place2.
   pathfilter(containsOnly(??loc1, yago:isLocatedIn) &&
```

```
   containsOnly(??loc2,yago:isLocatedIn))
}
```

Query 5

```
select ?person  ??loc
where {?person yago:isKnownFor ?smth.
   ?smth ??related yago:wordnet_physical_phenomenon>.
   ?place ??loc ?country.
   ?country rdf:type yago:wikicategory_European_countries.
   ?country rdf:type yago:wikicategory_Mediterranean.
   ?person yago:wasBornIn ?place.
   pathfilter(containsOnly(??loc,yago:isLocatedIn))
}
```

Queries for the UniProt dataset

Query 1

```
select ?a ?mod ??inf
where {?a <mnemonic> ?vo.
   ?a <replacedBy> <P62965>.
   ?a <type> <Protein>.
   ?a <modified> ?mod.
   ?b <modified> "2005-08-30".
   ?b <replacedBy> <P62964>.
   ?b <type> <Protein>.
   ?a <replacedBy> ?ab.
   ?ab ??inf ?b .
}
```

Query 2

```
select ?a ?vo
where {?a <mnemonic> ?vo.
   ?a <replacedBy> <P62965>.
   ?a <type> <Protein>.
   ?a <modified> "1990-11-01".
   ?a <replacedBy> <P62966>.
   ?a ??p ?b.
   ?b <modified> "2005-08-30".
   ?b <replacedBy> <P62964>.
   ?b <reviewed> "false".
   ?b <obsolete> "true".
   ?b <type> <Protein>}
```

Query 3

```
select ?a ?vo
```

```
where {?a <annotation> ?vo.
?a <seeAlso> <interpro/IPR000842>.
?a <annotation> <540A71>.
?a <seeAlso> <geneid/945772>.
?a <annotation> <540A7D>.
?a <citation> <9298646>.
?b <obsolete> "true".
?b <replacedBy> <P0A718>.
   ?b <reviewed> "true".
   ?b <mnemonic> "KPRS ECOLI".
   ?b <type> <Protein>.
   ?a ??p ?b }
```

APPENDIX B

Reachability queries

Queries for the YAGO2 dataset

All queries use the following prefix:
yago:<http://yago-knowledge.org/resource/>

Query 1

```
select ?country ?area where {
yago:Berlin yago:isLocatedIn* ?country.
?country yago:dealsWith ?area.
?area rdf:type yago:wikicategory_Member_states_of_NATO }
```

Query 2

```
select ?city ?b ?area where {
?city rdf:type yago:wikicategory_Capitals_in_Europe .
?city yago:isLocatedIn* ?b.
?b yago:dealsWith ?area }
```

Query 3

```
select ?a ?b ?area where {
?a yago:isLocatedIn* ?b.
?b yago:dealsWith ?area.
?a yago:isPreferredMeaningOf "Berlin"@eng}
```

Query 4

```
select ?a ?b ?type where {
?a yago:isPreferredMeaningOf "Berlin"@eng.
?a yago:isLocatedIn* ?b.
?b type ?type.
}
```

Query 5

```
select * where  {
```

```
?person yago:isMarriedTo* ?spouse.
?spouse yago:owns ?entity.
?entity yago:isLocatedIn* yago:United_States }
limit 100
```

Query 6

```
select * where  {
?airport1 a yago:wikicategory_Airports_in_the_Netherlands.
?airport1 yago:hasLongitude ?long.
?airport1 yago:hasLatitude ?lat.
?airport1 yago:isConnectedTo* ?place.
?place a yago:wikicategory_Mediterranean_port_cities_and_towns_in_Spain.
?place yago:wasCreatedOnDate ?date.
?place yago:hasNumberOfPeople ?people. }
limit 10
```

APPENDIX C

Cypher queries

MusicBrainz Query 1

```
MATCH (a:Artist)-[r]-(b)
WITH r,b
LIMIT 10000
RETURN type(r),labels(b),count(*)
ORDER BY count(*) desc
```

MusicBrainz Query 2

```
MATCH (a:Artist {name:'John Lennon'})-
  [r:MEMBER_OF_BAND*3..6]-(o:Person)
RETURN o.name,count(*)
ORDER BY count(*) DESC
LIMIT 10
```

MusicBrainz Query 3

```
MATCH (x:Country {name: {name_1}}),
  (y:Country {name: {name_2}}),
  (a:Artist)-[:FROM_AREA]-(x),
  (a:Artist)-[:RECORDING_CONTRACT]-(l:Label),
  (l)-[:FROM_AREA]-(y)
RETURN a,l,y,x
```

MusicBrainz Query 4

```
MATCH (a:Artist {name: 'John Lennon'})-
   [:CREDITED_AS]->(b)-
   [:CREDITED_ON]->(t:Track)
RETURN t.name
```

MusicBrainz Query 5

```
MATCH (a:Song)-[r]-(b)
WITH r,b
LIMIT 10000
RETURN type(r),labels(b),count(*)
```

```
ORDER BY count(*) DESC
```

MusicBrainz Query 6

```
MATCH (gb:Country {name:'United Kingdom'}),
   (usa:Country {name:'United States'}),
   (a:Artist)-[:FROM_AREA]-(gb),
   (a:Artist)-[:RECORDING_CONTRACT]-(l:Label),
   (l)-[:FROM_AREA]-(usa)
RETURN a,l,usa,gb
```

AccessControl Query 1

```
MATCH (admin:Administrator {name:{adminName}})
MATCH (admin)-[:MEMBER_OF]->(group)-[:ALLOWED_INHERIT]->
(company:Company)<-[:CHILD_OF*0..3]-(subcompany)<-
[:WORKS_FOR]-(employee)-[:HAS_ACCOUNT]->(account)
WHERE NOT ((admin)-[:MEMBER_OF]->()-[:DENIED]->()<-
[:CHILD_OF*0..3]-(subcompany))
RETURN account.name AS account
UNION MATCH (admin:Administrator {name:{adminName}})
MATCH (admin)-[:MEMBER_OF]->(group)-[:ALLOWED_DO_NOT_INHERIT]->
(company:Company)<-[:WORKS_FOR]-(employee)-[:HAS_ACCOUNT]->(account)
RETURN account.name AS account
```

AccessControl Query 2

```
MATCH (admin:Administrator {name:{adminName}})
MATCH (admin)-[:MEMBER_OF]->(group)-[:ALLOWED_INHERIT]->
(company:Company)<-[:WORKS_FOR]-(employee)-
[:HAS_ACCOUNT]-> (account)
WHERE NOT ((admin)-[:MEMBER_OF]->()-
 [:DENIED]->(company))
RETURN account.name AS account
UNION MATCH (admin:Administrator {name:{adminName}})
MATCH (admin)-[:MEMBER_OF]->(group)-[:ALLOWED_INHERIT]->
(company:Company) <-[:CHILD_OF]-(subcompany)<-
[:WORKS_FOR]-(employee)-[:HAS_ACCOUNT]->(account)
WHERE NOT ((admin)-[:MEMBER_OF]->()-[:DENIED]->()
 <-[:CHILD_OF]-(subcompany))
RETURN account.name AS account
UNION MATCH (admin:Administrator {name:{adminName}})
MATCH (admin)-[:MEMBER_OF]->(group)-
[:ALLOWED_DO_NOT_INHERIT]->(company:Company)
<-[:WORKS_FOR]-(employee)-[:HAS_ACCOUNT]->
(account)
RETURN account.name AS account
```

AccessControl Query 3

```
MATCH (admin:Administrator {name:{adminName}})
MATCH paths=(admin)-[:MEMBER_OF]->()-[:ALLOWED_INHERIT]->
(company)<-[:WORKS_FOR]-(employee)-
[:HAS_ACCOUNT]->(account)
WHERE NOT ((admin)-[:MEMBER_OF]->()-[:DENIED]->(company))
RETURN employee.name AS employee, account.name AS account
UNION MATCH (admin:Administrator {name:{adminName}})
MATCH paths=(admin)-[:MEMBER_OF]->()-[:ALLOWED_INHERIT]->
()<-[:CHILD_OF]-(company)<-[:WORKS_FOR]-(employee)-
[:HAS_ACCOUNT]->(account)
WHERE NOT ((admin)-[:MEMBER_OF]->()-[:DENIED]->
()<-[:CHILD_OF]-(company))
RETURN employee.name AS employee, account.name AS account
UNION MATCH (admin:Administrator {name:{adminName}})
MATCH paths=(admin)-[:MEMBER_OF]->()-[:ALLOWED_DO_NOT_INHERIT]->
()<-[:WORKS_FOR]-(employee)-[:HAS_ACCOUNT]->(account)
RETURN employee.name AS employee, account.name AS account
```

AccessControl Query 4

```
MATCH (admin:Administrator {name:{adminName}}),
(resource:Resource {name:{resourceName}})
MATCH p=(admin)-[:MEMBER_OF]->()-
[:ALLOWED_INHERIT]->()<-[:CHILD_OF*0..3]-(company)-
[:WORKS_FOR|HAS_ACCOUNT*1..2]-(resource)
WHERE NOT ((admin)-[:MEMBER_OF]->()-
[:DENIED]->()<-[:CHILD_OF*0..3]-(company))
RETURN count(p) AS accessCount
UNION MATCH (admin:Administrator {name:{adminName}}),
(resource:Resource {name:{resourceName}})
MATCH p=(admin)-[:MEMBER_OF]->()-
[:ALLOWED_DO_NOT_INHERIT]->(company)-
[:WORKS_FOR|HAS_ACCOUNT*1..2]-(resource)
RETURN count(p) AS accessCount
```

APPENDIX D

LDBC queries

Query 1

```
MATCH (:Person {id:{1}})-[path:KNOWS*1..3]-(friend:Person)
WHERE friend.firstName = {2}
WITH friend, min(length(path)) AS distance
ORDER BY distance ASC, friend.lastName ASC, friend.id ASC
LIMIT {3}
MATCH (friend)-[:IS_LOCATED_IN]->(friendCity:City)
OPTIONAL MATCH (friend)-[studyAt:STUDY_AT]->(uni:University)-[:IS_LOCATED_IN]->(uniCity:City)
WITH friend,
  collect(CASE uni.name
            WHEN null THEN null
            ELSE [uni.name, studyAt.classYear, uniCity.name]
          END) AS unis,
  friendCity,
  distance
OPTIONAL MATCH (friend)-[worksAt:WORKS_AT]->
  (company:Company)-[:IS_LOCATED_IN]->(companyCountry:Country)
WITH friend,
  collect(CASE company.name
            WHEN null THEN null
            ELSE [company.name, worksAt.workFrom, companyCountry.name]
          END) AS companies,
  unis,
  friendCity,
  distance
RETURN
  friend.id AS id,
  friend.lastName AS lastName,
  distance,
  friend.birthday AS birthday,
  friend.creationDate AS creationDate,
  friend.gender AS gender,
  friend.browserUsed AS browser,
  friend.locationIP AS locationIp,
  friend.email AS emails,
  friend.languages AS languages,
  friendCity.name AS cityName,
```

```
  unis,
  companies
ORDER BY distance ASC, friend.lastName ASC, friend.id ASC
LIMIT {3}
```

Query 2

```
MATCH (:Person {id:{1}})-[:KNOWS]-(friend:Person)<-[:HAS_CREATOR]-(message)
WHERE message.creationDate <= {2} AND (message:Post OR message:Comment)
RETURN
  friend.id AS personId,
  friend.firstName AS personFirstName,
  friend.lastName AS personLastName,
  message.id AS messageId,
  CASE has(message.content)
    WHEN true THEN message.content
    ELSE message.imageFile
  END AS messageContent,
  message.creationDate AS messageDate
ORDER BY messageDate DESC, messageId ASC
LIMIT {3}
```

Query 3

```
MATCH (person:Person {id:{1}})-[:KNOWS*1..2]-(friend:Person)<-[:HAS_CREATOR]-(messageX),
      (messageX)-[:IS_LOCATED_IN]->(countryX:Country)
WHERE not(person=friend) AND
  not((friend)-[:IS_LOCATED_IN]->()-[:IS_PART_OF]->(countryX))
  AND countryX.name={2}
  AND messageX.creationDate>={4}
  AND messageX.creationDate<{5}
WITH friend, count(DISTINCT messageX) AS xCount
MATCH (friend)<-[:HAS_CREATOR]-(messageY)-[:IS_LOCATED_IN]->(countryY:Country)
WHERE countryY.name={3}
  AND not((friend)-[:IS_LOCATED_IN]->()-[:IS_PART_OF]->(countryY))
  AND messageY.creationDate>={4}
  AND messageY.creationDate<{5}
WITH friend.id AS friendId,
  friend.firstName AS friendFirstName,
  friend.lastName AS friendLastName,
  xCount,
  count(DISTINCT messageY) AS yCount
RETURN
  friendId,
  friendFirstName,
  friendLastName,
  xCount,
  yCount,
  xCount + yCount AS xyCount
```

```
ORDER BY xyCount DESC, friendId ASC
LIMIT {6}
```

Query 4

```
MATCH (person:Person {id:{1}})-[:KNOWS]-(:Person)<-[:HAS_CREATOR]-(post:Post)
  -[HAS_TAG]->(tag:Tag)
WHERE post.creationDate >= {2} AND post.creationDate < {3}
OPTIONAL MATCH (tag)<-[:HAS_TAG]-(oldPost:Post)
WHERE oldPost.creationDate < {2}
WITH tag, post, length(collect(oldPost)) AS oldPostCount
WHERE oldPostCount=0
RETURN
  tag.name AS tagName,
  length(collect(post)) AS postCount
ORDER BY postCount DESC, tagName ASC
LIMIT {4}
```

Query 5

```
MATCH (person:Person {id:{1}})-[:KNOWS*1..2]-(friend:Person)
   <-[membership:HAS_MEMBER]-(forum:Forum)
WHERE membership.joinDate>{2} AND not(person=friend)
WITH DISTINCT friend, forum
OPTIONAL MATCH (friend)<-[:HAS_CREATOR]-(post:Post)<-[:CONTAINER_OF]-(forum)
WITH forum, count(post) AS postCount
RETURN
  forum.title AS forumName,
  postCount
ORDER BY postCount DESC, forum.id ASC
LIMIT {3}
```

Query 6

```
MATCH (person:Person {id:{1}})-[:KNOWS*1..2]-(friend:Person),
      (friend)<-[:HAS_CREATOR]-(friendPost:Post)-[:HAS_TAG]->(knownTag:Tag {name:{2}})
WHERE not(person=friend)
MATCH (friendPost)-[:HAS_TAG]->(commonTag:Tag)
WHERE not(commonTag=knownTag)
WITH DISTINCT commonTag, knownTag, friend
MATCH (commonTag)<-[:HAS_TAG]-(commonPost:Post)-[:HAS_TAG]->(knownTag)
WHERE (commonPost)-[:HAS_CREATOR]->(friend)
RETURN
  commonTag.name AS tagName,
  count(commonPost) AS postCount
ORDER BY postCount DESC, tagName ASC
LIMIT {3}
```

Query 7

```
MATCH (person:Person {id:{1}})<-[:HAS_CREATOR]-(message)<-[like:LIKES]-(liker:Person)
WITH liker, message, like.creationDate AS likeTime, person
ORDER BY likeTime DESC, message.id ASC
WITH liker, head(collect({msg: message, likeTime: likeTime})) AS latestLike, person
RETURN
  liker.id AS personId,
  liker.firstName AS personFirstName,
  liker.lastName AS personLastName,
  latestLike.likeTime AS likeTime,
  not((liker)-[:KNOWS]-(person)) AS isNew,
  latestLike.msg.id AS messageId,
  latestLike.msg.content AS messageContent,
  latestLike.likeTime - latestLike.msg.creationDate AS latencyAsMilli
ORDER BY likeTime DESC, personId ASC
LIMIT {2}
```

Query 8

```
MATCH (start:Person {id:{1}})<-[:HAS_CREATOR]-()<-[:REPLY_OF]-
  (comment:Comment)-[:HAS_CREATOR]->(person:Person)
RETURN
  person.id AS personId,
  person.firstName AS personFirstName,
  person.lastName AS personLastName,
  comment.id AS commentId,
  comment.creationDate AS commentCreationDate,
  comment.content AS commentContent
ORDER BY commentCreationDate DESC, commentId ASC
LIMIT {2}
```

Query 9

```
MATCH (:Person {id:{1}})-[:KNOWS*1..2]-(friend:Person)<-[:HAS_CREATOR]-(message)
WHERE message.creationDate < {2}
RETURN DISTINCT
  message.id AS messageId,
  CASE has(message.content)
    WHEN true THEN message.content
    ELSE message.imageFile
  END AS messageContent,
  message.creationDate AS messageCreationDate,
  friend.id AS personId,
  friend.firstName AS personFirstName,
  friend.lastName AS personLastName
ORDER BY message.creationDate DESC, message.id ASC
LIMIT {3}
```

Query 10

```
MATCH (person:Person {id:{1}})-[:KNOWS*2..2]-(friend:Person)-[:IS_LOCATED_IN]->(city:City)
WHERE ((friend.birthday_month = {2} AND friend.birthday_day >= 21)
      OR (friend.birthday_month = ({2}+1)%12 AND friend.birthday_day < 22))
      AND not(friend=person)
      AND not((friend)-[:KNOWS]-(person))
WITH DISTINCT friend, city, person
OPTIONAL MATCH (friend)<-[:HAS_CREATOR]-(post:Post)
WITH friend, city, collect(post) AS posts, person
WITH
  friend,
  city,
  length(posts) AS postCount,
  length([p IN posts WHERE (p)-[:HAS_TAG]->(:Tag)<-[:HAS_INTEREST]-(person)])
  AS commonPostCount
RETURN
  friend.id AS personId,
  friend.firstName AS personFirstName,
  friend.lastName AS personLastName,
  friend.gender AS personGender,
  city.name AS personCityName,
  commonPostCount - (postCount - commonPostCount) AS commonInterestScore
ORDER BY commonInterestScore DESC, personId ASC
LIMIT {4}
```

Query 11

```
MATCH (person:Person {id:{1}})-[:KNOWS*1..2]-(friend:Person)
WHERE not(person=friend)
WITH DISTINCT friend
MATCH (friend)-[worksAt:WORKS_AT]->(company:Company)-
  [:IS_LOCATED_IN]->(:Country {name:{3}})
WHERE worksAt.workFrom < {2}
RETURN
  friend.id AS friendId,
  friend.firstName AS friendFirstName,
  friend.lastName AS friendLastName,
  worksAt.workFrom AS workFromYear,
  company.name AS companyName
ORDER BY workFromYear ASC, friendId ASC, companyName DESC
LIMIT {4}
```

Query 12

```
MATCH (:Person {id:{1}})-[:KNOWS]-(friend:Person)
OPTIONAL MATCH (friend)<-[:HAS_CREATOR]-(comment:Comment)-
       [:REPLY_OF]->(:Post)-[:HAS_TAG]->(tag:Tag),
        (tag)-[:HAS_TYPE]->(tagClass:TagClass)-
```

```
        [:IS_SUBCLASS_OF*0..]->(baseTagClass:TagClass)
WHERE tagClass.name = {2} OR baseTagClass.name = {2}
RETURN
  friend.id AS friendId,
  friend.firstName AS friendFirstName,
  friend.lastName AS friendLastName,
  collect(DISTINCT tag.name) AS tagNames,
  count(DISTINCT comment) AS count
ORDER BY count DESC, friendId ASC
LIMIT {3}
```

Query 13

```
MATCH (person1:Person {id:{1}}), (person2:Person {id:{2}})
OPTIONAL MATCH path = shortestPath((person1)-[:KNOWS]-(person2))
RETURN CASE path IS NULL
         WHEN true THEN -1
         ELSE length(path)
       END AS pathLength
```

References

[1] Charu C. Aggarwal and Haixun Wang. *Managing and Mining Graph Data*. Springer, 2010.

[2] R. Agrawal, A. Borgida, and H. V. Jagadish. "Efficient Management of Transitive Relationships in Large Data and Knowledge Bases". In: *SIGMOD Rec.* 18.2 (June 1989), pp. 253–262.

[3] R. Agrawal and H.V. Jagadish. "Algorithms for searching massive graphs". In: *Knowledge and Data Engineering, IEEE Transactions on* 6.2 (1994), pp. 225–238.

[4] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. "DBXplorer: A System for Keyword-Based Search over Relational Databases". In: ICDE '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 5–.

[5] N. Alon, R. Yuster, and U. Zwick. "Finding and counting given length cycles". In: *Algorithmica* 17.3 (1997), pp. 209–223.

[6] Omid Amini, Fedor V. Fomin, and Saket Saurabh. "Counting Subgraphs via Homomorphisms". In: Lecture Notes in Computer Science 5555 (2009). Ed. by Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris Nikoletseas, and Wolfgang Thomas, pp. 71–82.

[7] Kemafor Anyanwu, Angela Maduko, and Amit Sheth. "SPARQ2L: Towards Support for Subgraph Extraction Queries in RDF Databases". In: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada: ACM, 2007, pp. 797–806.

[8] Kemafor Anyanwu and Amit Sheth. "ρ-Queries: Enabling Querying for Semantic Associations on the Semantic Web". In: *Proceedings of the 12th International Conference on World Wide Web*. WWW '03. Budapest, Hungary: ACM, 2003, pp. 690–699.

[9] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. "Counting Beyond a Yottabyte, or How SPARQL 1.1 Property Paths Will Prevent Adoption of the Standard". In: *Proceedings of the 21st International Conference on World Wide Web*. WWW '12. Lyon, France: ACM, 2012, pp. 629–638.

[10] Marcelo Arenas, Claudio Gutierrez, Daniel P. Miranker, Jorge Pérez, and Juan F. Sequeda. "Querying Semantic Data on the Web?" In: *SIGMOD Rec.* 41.4 (Jan. 2013), pp. 6–17.

[11] Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. "An Empirical Study of Real-World SPARQL Queries". In: *CoRR* abs/1103.5043 (2011). URL: `http://arxiv.org/abs/1103.5043`.

[12] Pablo Barceló, Leonid Libkin, Anthony W. Lin, and Peter T. Wood. "Expressive Languages for Path Queries over Graph-Structured Data". In: *ACM Trans. Database Syst.* 37.4 (Dec. 2012), 31:1–31:46.

[13] Hannah Bast. "Car or Public Transport – Two Worlds". In: Lecture Notes in Computer Science 5760/2009. Springer, 2009, pp. 355–367.

[14] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. "In Transit to Constant Time Shortest-Path Queries in Road Networks". In: *ALENEX'07: Proceedings of the 2007 SIAM Workshop on Algorithm Engineering and Experiments*. SIAM. 2007.

[15] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. "Keyword Searching and Browsing in Databases using BANKS." In: *ICDE*. 2002, pp. 431–440.

[16] Peter Boncz, Thomas Neumann, and Orri Erling. "TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark". In: *TPCTC*. 2013.

[17] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. "Building an Efficient RDF Store over a Relational Database". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: ACM, 2013, pp. 121–132.

[18] Matthias Brcheler, Andrea Pugliese, and V.S. Subrahmanian. "DOGMA: A Disk-Oriented Graph Matching Algorithm for RDF Databases". In: *The Semantic Web - ISWC 2009*. Ed. by Abraham Bernstein, DavidR. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan. Vol. 5823. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 97–113.

[19] Miroslav Chlebk and Janka Chlebkov. "The Steiner tree problem on graphs: Inapproximability results". In: *Theoretical Computer Science* 406.3 (2008), pp. 207 –214. URL: `http://www.sciencedirect.com/science/article/pii/S0304397508004660`.

[20] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. "Reachability and Distance Queries via 2-Hop-Labels". In: *SODA'2002: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*. 2002, pp. 937–946.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd. MIT Press, 2009.

[22] Atish Das Sarma, Sreenivas Gollapudi, Marc Najork, and Rina Panigrahy. "A Sketch-Based Distance Oracle for Web-Scale Graphs". In: *WSDM'10: Proceedings of the 3rd ACM International Conference on Web Search and Data Mining*. New York, NY, USA: ACM, 2010, pp. 401–410.

[23] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271.

[24] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. "Finding Top-k Min-Cost Connected Trees in Databases." In: *ICDE'07*. 2007, pp. 836–845.

[25] S. E. Dreyfus and R. A. Wagner. "The steiner problem in graphs". In: *Networks* 1.3 (1971), pp. 195–207.

[26] Martin E. Dyer, Alan M. Frieze, and Mark Jerrum. "Approximately Counting Hamilton Paths and Cycles in Dense Graphs." In: *SIAM J. Comput.* 27.5 (1998), pp. 1262–1272.

[27] Orri Erling. "Virtuoso, a Hybrid RDBMS/Graph Column Store". In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 3–8.

[28] Orri Erling and Ivan Mikhailov. "Virtuoso: RDF Support in a Native RDBMS". In: *Semantic Web Information Management - A Model-Based Perspective*. 2009, pp. 501–519.

[29] Leonidas Fegaras. "A New Heuristic for Optimizing Large Queries". In: *In 9th International Conference, DEXA'98*. Springer-Verlag, 1998, pp. 726–735.

[30] César Galindo-Legaria and Milind Joshi. "Orthogonal Optimization of Subqueries and Aggregation". In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. SIGMOD '01. Santa Barbara, California, USA: ACM, 2001, pp. 571–581.

[31] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. "Reach for A*: Efficient Point-to-Point Shortest Path Algorithms". In: *ALENEX'06: Proceedings of the 2006 SIAM Workshop on Algorithm Engineering and Experiments*. SIAM. 2006.

[32] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. "Reach for A*: Efficient point-to-point shortest path algorithms". In: *Workshop on Algorithm Engineering & Experiments*. 2006, pp. 129–143.

[33] Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. "Keyword proximity search in complex data graphs". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 927–940.

[34] Andrey Gubichev. "Analysis and Classification of Choke Points". In: `http://ldbc.eu/sites/default/files/D2.2.1_final_v2.pdf`.

[35] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. "Sparqling kleene: fast property paths in RDF-3X". In: *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*. 2013, p. 14.

[36] Andrey Gubichev and Peter Boncz. "Parameter Curation For Benchmark Queries". en. In: *Proceedings of the TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC, 2014)*. Ed. by M. Poess and R. Niambar. 2014, pp. –.

[37] Andrey Gubichev and Thomas Neumann. "Exploiting the query structure for efficient join ordering in SPARQL queries". In: *Proc. 17th International Conference on Extending Database Technology (EDBT), Athens, Greece, March 24-28, 2014*. Ed. by Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy. OpenProceedings.org, 2014, pp. 439–450.

[38] Andrey Gubichev and Thomas Neumann. "Fast approximation of steiner trees in large graphs". In: *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*. 2012, pp. 1497–1501.

[39] Andrey Gubichev and Thomas Neumann. "Path Query Processing on Very Large RDF Graphs". In: *Proceedings of the 14th International Workshop on the Web and Databases 2011, WebDB 2011, Athens, Greece, June 12, 2011*. 2011.

[40] Andrey Gubichev, Srikanta J. Bedathur, Stephan Seufert, and Gerhard Weikum. "Fast and accurate estimation of shortest paths in large graphs". In: *Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010, Toronto, Ontario, Canada, October 26-30, 2010*. 2010, pp. 499–508.

[41] Olaf Hartig. "Reconciliation of RDF* and Property Graphs". In: *CoRR* abs/1409.3288 (2014). URL: `http://arxiv.org/abs/1409.3288`.

[42] Olaf Hartig and Bryan Thompson. "Foundations of an Alternative Approach to Reification in RDF". In: *CoRR* abs/1406.3399 (2014). URL: `http://arxiv.org/abs/1406.3399`.

[43] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. "BLINKS: ranked keyword searches on graphs". In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. SIGMOD '07. Beijing, China: ACM, 2007, pp. 305–316. ISBN: 978-1-59593-686-8.

[44] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2011.

[45] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. "YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia". In: *Artif. Intell.* 194 (Jan. 2013), pp. 28–61. ISSN: 0004-3702.

[46] Vagelis Hristidis and Yannis Papakonstantinou. "DISCOVER: Keyword Search in Relational Databases". In: *VLDB*. 2002, pp. 670–681.

[47] Maciej Janik and Krys Kochut. "BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery". In: *The Semantic Web ISWC 2005*. Ed. by Yolanda Gil, Enrico Motta, V.Richard Benjamins, and MarkA. Musen. Vol. 3729. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 431–445.

[48] Jena. In: URL: `http://jena.sourceforge.net/`.

[49] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. "3-HOP: A High-Compression Indexing Scheme for Reachability Query". In: *SIGMOD'09: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. ACM, 2009, pp. 813–826.

[50] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. "Efficiently Answering Reachability Queries on Very Large Directed Graphs". In: *SIGMOD'08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. ACM, 2008, pp. 595–608.

[51] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. "Bidirectional expansion for keyword search on graph databases". In: *Proceedings of the 31st international conference on Very large data bases*. VLDB '05. Trondheim, Norway: VLDB Endowment, 2005, pp. 505–516.

[52] Mehdi Kargar and Aijun An. "Keyword search in graphs: finding r-cliques". In: *Proc. VLDB Endow.* 4.10 (July 2011), pp. 681–692.

[53] Richard M. Karp. "Reducibility among combinatorial problems". In: *Complexity of Computer Computations*. Ed. by R. E. Miller and J. W. Thatcher. New York: Plenum, 1972, pp. 85–103.

[54] Gjergji Kasneci, Maya Ramanath, Mauro Sozio, Fabian M. Suchanek, and Gerhard Weikum. "STAR: Steiner-Tree Approximation in Relationship Graphs." In: *ICDE*. Ed. by Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng. IEEE, 2009, pp. 868–879.

[55] Gjergji Kasneci, Maya Ramanath, Fabian Suchanek, and Gerhard Weikum. "The YAGO-NAGA Approach to Knowledge Discovery". In: *SIGMOD Rec.* 37.4 (Mar. 2009), pp. 41–47.

[56] Donald E. Knuth. *Seminumerical Algorithms*. The Art of Computer Programming. Addison-Wesley, 1981.

[57] Krys J. Kochut and Maciej Janik. "SPARQLeR: Extended Sparql for Semantic Association Discovery". In: ESWC '07. Innsbruck, Austria: Springer-Verlag, 2007, pp. 145–159.

[58] Lawrence T. Kou, George Markowsky, and Leonard Berman. "A Fast Algorithm for Steiner Trees". In: *Acta Inf.* 15 (1981), pp. 141–145.

[59] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. "What is Twitter, a social network or a news media?" In: *WWW '10: Proceedings of the 19th international conference on World wide web*. Raleigh, North Carolina, USA: ACM, 2010, pp. 591–600.

[60] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. "Predicting Positive and Negative Links in Online Social Networks". In: *WWW'10: Proceedings of the 19th International World Wide Web Conference*. ACM. 2010, pp. 641–650.

[61] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. "Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations". In: *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. KDD '05. Chicago, Illinois, USA: ACM, 2005, pp. 177–187.

[62] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. "Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations". In: *KDD'2005: Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2005, pp. 177–187.

[63] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters". In: *arXiv:0810.1355v1* (2008).

[64] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. "EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 903–914.

[65] Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. "Effective keyword search in relational databases". In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. SIGMOD '06. Chicago, IL, USA: ACM, 2006, pp. 563–574.

[66] Katja Losemann and Wim Martens. "The Complexity of Evaluating Path Expressions in SPARQL". In: *Proceedings of the 31st Symposium on Principles of Database Systems*. PODS '12. Scottsdale, Arizona, USA: ACM, 2012, pp. 101–112.

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

[68] Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. "Spark: top-k keyword query in relational databases". In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. SIGMOD '07. Beijing, China: ACM, 2007, pp. 115–126.

[69] Volker Markl, Peter J. Haas, Marcel Kutsch, Nimrod Megiddo, Utkarsh Srivastava, and Tam Minh Tran. "Consistent selectivity estimation via maximum entropy". In: *VLDB J.* 16.1 (2007), pp. 55–76.

[70] Pham Minh Duc. "Self-organizing structured RDF in MonetDB". In: *Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on*. 2013, pp. 310–313.

[71] Pham Minh Duc, Peter Boncz, and Orri Erling. "S3G2: A Scalable Structure-Correlated Social Graph Generator". en. In: *TPCTC*. 2012.

[72] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. "Measurement and Analysis of Online Social Networks". In: *SIGCOMM'07: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*. ACM. 2007, pp. 29–42.

[73] Guido Moerkotte. "Building Query Compilers". In: `http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf`.

[74] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. "DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data". In: *ISWC 2011*. 2011.

[75] Thomas Neumann. "Query Simplification: Graceful Degradation for Join-order Optimization". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Providence, Rhode Island, USA: ACM, 2009, pp. 403–414.

[76] Thomas Neumann and Guido Moerkotte. "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins". In: *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. 2011, pp. 984–994.

[77] 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. Providence, Rhode Island, USA: ACM, 2009, pp. 627–640.

[78] Thomas Neumann and Gerhard Weikum. "The RDF-3X Engine for Scalable Management of RDF Data". In: *The VLDB Journal* 19.1 (Feb. 2010), pp. 91–113.

[79] Meikel Poess and John M. Stephens Jr. "Generating thousand benchmark queries in seconds". In: VLDB. Toronto, Canada, 2004, pp. 1045–1053.

[80] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. "Fast Shortest Path Distance Estimation in Large Networks". In: *CIKM'09: Proceedings of the 18th ACM Conference on Information and Knowledge Management*. ACM, 2009, pp. 867–876.

[81] Arnau Prat and Alex Averbuch. "Benchmark design for navigational pattern matching benchmarking". In: `http://ldbc.eu/sites/default/files/LDBC_D3.3.34.pdf`.

[82] Josep M. Pujol, Georgos Siganos, Vijay Erramilli, and Pablo Rodriguez. "Scaling Online Social Networks without Pains". In: *NetDB'09: 5th International Workshop on Networking Meets Databases*. 2009.

[83] Miao Qiao, Hong Cheng, and Jeffrey Xu Yu. "Querying shortest path distance with bounded errors in large graphs". In: *Proceedings of the 23rd international conference on Scientific and statistical database management*. SSDBM'11. Portland, OR: Springer-Verlag, 2011, pp. 255–273.

[84] Miao Qiao, Hong Cheng, Lijun Chang, and Jeffrey Xu Yu. "Approximate Shortest Distance Computing: a query-dependent local landmark scheme". In: *Proceedings of the 18th International Conference on Data Engineering*. ICDE '12. Washington, DC, USA: IEEE Computer Society, 2012.

[85] Ben Roberts and Dirk P. Kroese. "Estimating the Number of s-t Paths in a Graph". In: *J. Graph Algorithms Appl.* 11.1 (2007), pp. 195–214.

[86] Gabriel Robins and Alexander Zelikovsky. "Tighter Bounds for Graph Steiner Tree Approximation". In: *SIAM J. Discret. Math.* 19.1 (May 2005), pp. 122–134.

[87] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, 2013.

[88] Satya Sanket Sahoo. "Semantic Provenance: Modeling, Querying, and Application in Scientific Discovery". In: PhD Thesis. 2010.

[89] Jagnan Sankaranarayanan and Hanan Samet. "Distance Oracles for Spatial Networks". In: *ICDE'09: Proceedings of the 2009 IEEE International Conference on Data Engineering*. Shanghai, China: IEEE Computer Society, 2009, pp. 652–663.

[90] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. "HOPI: An Efficient Connection Index for Complex XML Document Collections". In: *EDBT'04: Proceedings of the 9th International Conference on Extending Database Technology*. Lecture Notes in Computer Science 2992. 2004, pp. 237–255.

[91] C.-P. Schnorr. "An algorithm for transitive closure with linear expected time". In: *Theoretical Computer Science*. Vol. 48. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1977, pp. 329–338.

[92] S. Seufert, A. Anand, S. Bedathur, and G. Weikum. "FERRARI: Flexible and efficient reachability range assignment for graph indexing". In: *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. 2013, pp. 1009–1020.

[93] Christian Sommer, Elad Verbin, and Wei Yu. "Distance Oracles for Sparse Graphs". In: *FOCS'09: Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science*. Atlanta, GA, United States, 2009, pp. 703–712.

[94] John M. Stephens and Meikel Poess. "MUDD: a multi-dimensional data generator". In: *SIGSOFT Softw. Eng. Notes* 29.1 (Jan. 2004), pp. 104–109.

[95] 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. Beijing, China: ACM, 2008, pp. 595–604.

[96] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. "YAGO: A Large Ontology from Wikipedia and WordNet". In: *J. Web Sem.* 6.3 (2008), pp. 203–217.

[97] Mikkel Thorup and Uri Zwick. "Approximate Distance Oracles". In: *STOC'01: Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*. ACM. 2001, pp. 183–192.

[98] Thanh Tran, Haofen Wang, Sebastian Rudolph, and Phulipp Cimiano. "Top-k Exploration of Query Candidates for Efficient Keyword Search on Graph-Shaped (RDF) Data". In: *ICDE'09: Proceedings of the 2009 IEEE International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 405–416. DOI: 10.1109/ICDE.2009.119.

[99] Silke Trißl and Ulf Leser. "Fast and Practical Indexing and Querying of Very Large Graphs". In: *SIGMOD'07: Proceedings of the 2007 ACM SIGMOD Intl. Conf. on Management of Data*. ACM. 2007, pp. 845–856.

[100] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter Boncz. "Heuristics-based Query Optimisation for SPARQL". In: *Proceedings of the 15th International Conference on Extending Database Technology*. EDBT '12. Berlin, Germany: ACM, 2012, pp. 324–335.

[101] 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. Vancouver, British Columbia, Canada: AAAI Press, 2007, pp. 1465–1470.

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

[103] Duncan J. Watts and Steven H. Strogatz. "Collective dynamics of /'small-world/' networks". In: *Nature* 393.6684 (June 1998), pp. 440–442.

[104] Mohamed Yahya, Klaus Berberich, Shady Elbassuoni, Maya Ramanath, Volker Tresp, and Gerhard Weikum. "Natural Language Questions for the Web of Data". In: EMNLP-CoNLL. Jeju Island, Korea: Association for Computational Linguistics, 2012, pp. 379–390.

[105] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. "Keyword Search in Relational Databases: A Survey". In: *IEEE Data Eng. Bull.* 33.1 (2010), pp. 67–78.

[106] Uri Zwick. "Exact and Approximate Distances in Graphs – A Survey". In: *ESA'01: Procceding of the 9th Annual European Symposium on Algorithms*. Lecture Notes in Computer Science 2161/2001. Springer, 2001.