Eindhoven University of Technology

MASTER

Telepath
a path-index based graph database engine

Mak, G.

Award date:
2017

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Department of Mathematics and Computer Science
Databases Research Group

Telepath: A path-index based graph database engine

MSc Thesis

Giedo Mak

Supervisors:
dr. G.H.L. Fletcher
dr. N. Yakovets

External Supervisor:
prof. dr. A. Poulovassilis
(Birkbeck University of London)

Assessment Committee:
dr. G.H.L. Fletcher
dr. V. Menkovski
prof. dr. A. Poulovassilis
dr. N. Yakovets

Eindhoven, November 2017

No rest for the wicked.

— Mr. Robot

Dedicated to my parents.

Abstract

Massive graph-structured data collections are omnipresent in modern data management scenarios such as social networks, linked open data, and chemical compound databases. Regular path queries for graph databases are eminently useful. However, these queries are difficult to optimize to evaluate efficiently.

We have embarked on a project we call *Telepath* to design and engineer a path-index based graph database engine. Recent previous work on *path* indexing showed results which yield, on average, orders of magnitude faster query processing times in comparison with `Neo4j`, a popular open-source native graph database.

Our work presents an end-to-end solution to evaluate a regular path query efficiently. The core of our end-to-end solution is the dynamic programming based query planner. The query planner chooses a physical plan in a bottom-up fashion by using cost estimates. Chosen physical plans for smaller subproblems are used to construct the physical plan for the global problem. Cost estimates are provided by using cardinality estimates for intermediate results. The cardinality estimates are produced through relation statistics maintained over the graph data. Switching to the end of the life-of-a-query, a *path* index is used to retrieve (intermediate) query results.

The main contributions can be summed up into the following four main areas: relation statistics based cardinality estimation where we provide a novel implementation for an existing concept, dynamic programming based query planning where we provide a novel implementation for an existing concept, query evaluation using a disk-based *path* index where we use an existing implementation of a *path* index. However, we provide our own concept and implementation for end-to-end query evaluation while leveraging a *path* index. We designed, engineered, and tested the proposed approach.

Using one synthetic and one real-world dataset, we provide an empirical evaluation of the cardinality estimation, query planning, and query evaluation phases with a *path* index available for paths of length one and two. Our relation statistics based cardinality estimator generates estimates with a mean error of **0.56** on a scale from zero to one. This enables the dynamic programming based query planner to always choose a near optimal physical plan, as shown by a mean error of **0.02** during our experiments. Ultimately, we achieve a **9x** mean speedup for query evaluation times compared to `Neo4j`, a current state of the art graph database engine.

We see that *Telepath* performs particularly well for smaller result sets. We achieve a mean speedup of **29x** for queries which evaluate to small result sets of $0 - 100k$ tuples. We see that the evaluation into larger result sets, and thus more intermediate results, is not able to leverage the disk-based *path* index in the same order of performance gain.

Our *Telepath* prototype illustrates the performance gains that can be obtained in comparison to a current state of the art graph database engine, and it acts as

a foundation for further research in the area of path-index based graph database engines due to its modular and generic design.

The source code of *Telepath* has been made publicly available to contribute to the open-source community.

Acknowledgement

My project has been carried out at the Eindhoven University of Technology in the Databases Research group from the Department of Mathematics and Computer Science. My weekly supervisors in Eindhoven have been dr. George Fletcher and dr. Nikolay Yakovets. External supervision came from prof. dr. Alexandra Poulovassilis who is allied to the Birkbeck University of London.

I would like to thank dr. George Fletcher for giving me the opportunity to carry out a project in the new Databases Research group. I would also like to thank him and dr. Nikolay Yakovets for their continued support over the last months guiding, supporting and advising me. I would also like to thank dr. Alexandra Poulovassilis for her committed involvement throughout the project and the enlightening Skype-sessions we had. I would also like to thank dr. Vlado Menkovski for serving on my assessment committee.

A special thanks goes to my parents, Ruud Mak and Astrid Marijs, who have been supporting and loving me unconditionally throughout my entire life. Another special thanks goes to Tessa van Amelsvoort, without whose love, humor and encouragement I would not have finished this thesis in time. Additionally, I would like to thank my friends who have supported me, including Max Sumrall for his helping hand throughout the project, Vincent Looijen for the occasional consumption of schnapps, Jim van Boven for blackmailing me with karting credits, Joep Sommers for the distressing black coffees in MetaForum, Pieter Ottink for the never-ending locker room talk, Stef Schenkelaars for the promising opportunities we've seized and are seizing, and to all my other friends who deserve many thanks for making my time in Eindhoven an unforgettable experience.

Thank you.

Giedo Mak

Contents

Contents ix

List of Figures xi

List of Tables xiii

Listings xv

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Goal . . . 2
- 1.3 Contributions . . . 2

2 Preliminaries **5**
- 2.1 Graphs . . . 5
- 2.2 Paths . . . 6
- 2.3 Path queries . . . 6
- 2.4 k-path index . . . 7

3 Related work **9**
- 3.1 Query optimization . . . 9
- 3.2 Cardinality estimation . . . 10
  - 3.2.1 Histograms . . . 10
  - 3.2.2 Relation statistics . . . 10
- 3.3 Path query evaluation . . . 11

4 The design and engineering of *Telepath* **13**
- 4.1 Architecture . . . 13
  - 4.1.1 Engineering . . . 14
- 4.2 Life of a Query . . . 16
  - 4.2.1 Query input . . . 16
  - 4.2.2 Parse the input . . . 16
  - 4.2.3 Physical plan selection . . . 17
  - 4.2.4 Evaluate the physical plan . . . 17
- 4.3 Query planning . . . 18
  - 4.3.1 Flatten the logical plan into a multi-children tree . . . 18
  - 4.3.2 Generate subtrees of a given size . . . 18
  - 4.3.3 Check containment of subtrees . . . 20
  - 4.3.4 Enumerate physical operators . . . 21
  - 4.3.5 Costing physical plans . . . 21
- 4.4 Status of engineering . . . 22

5 Dynamic programming based query planner **25**
- 5.1 Introduction . . . 25
- 5.2 Subproblems . . . 27
- 5.3 `P1`: Subexpressions of a given size . . . 28

5.4 P2, P3: Check containment of expression 29
5.5 P4: Physical operator enumeration 29
5.6 P5: Cost estimation 31
5.6.1 Cardinality estimation 32
5.6.2 Cost model 33
5.7 Physical plan selection 34
5.8 Size of the physical plan space 35

6 Experiments and benchmarking **39**
6.1 Experiment goals 39
6.2 Experiment setup 40
6.2.1 LUBM dataset 40
6.2.2 Advogato dataset 41
6.2.3 Dataset size 42
6.3 Experiment results 43
6.3.1 Exp1: Cardinality estimation evaluation 43
6.3.2 Exp2: Query planning evaluation 44
6.3.3 Exp3: Query execution evaluation 45
6.4 Discussion 49

7 Conclusions **51**
7.1 Future work 52

Bibliography **55**

Appendix **59**

A Regular path query grammar **59**

B Benchmark queries **61**
B.1 LUBM dataset queries 61
B.1.1 Regular path queries 61
B.1.2 Cypher queries 61
B.2 Advogato dataset queries 62
B.2.1 Regular path queries 62
B.2.2 Cypher queries 63

C Engineering Telepath **64**
C.1 Directory structure 64
C.2 Interface example 65
C.3 Documentation 66
C.3.1 Repository readme 66
C.3.2 Repository contributing guide 66
C.3.3 Query planner guide 66
C.3.4 Guide on adding an extra physical operator 66

List of Figures

2.1 A graph G_{ex} over vocabulary $\mathcal{L}$ = { associatedWith, enrolledAt, friendOf, parentOf }. 5

3.1 Query optimization architecture as described in [12]. 9

4.1 Schematic overview of the *Telepath* architecture. 15

5.1 Dynamic programming algorithm DPsize as described in [18]. . . 27
5.2 Schematic illustration of a join set as presented in [29]. 32
5.3 Schematic overview of the relation statistics maintained by the SYNOPSIS as presented in [29]. 32
5.4 Formula to estimate the cardinality of *path queries* as seen in [29]. 33

6.1 Formula to measure the quality of a cardinality estimation within the balanced range $[-1, 1]$ as seen in [25]. 43
6.2 Schematic illustration of how we compare the number of intermediate results of the chosen physical plan, $plan$, with the whole space of physical plans, $[plan_1, \ldots, plan_n]$. 45
6.3 Speedup of query evaluation in *Telepath* compared to Neo4j for the LUBM dataset plotted against the size of the query result set. 47
6.4 Speedup of query evaluation in *Telepath* compared to Neo4j for the Advogato dataset plotted against the size of the query result set. 47

C.1 The readme of the *Telepath* repository page 1. 67
C.2 The readme of the *Telepath* repository page 2. 68
C.3 The readme of the *Telepath* repository page 3. 69
C.4 The contributing guide of the *Telepath* repository page 1. 70
C.5 The contributing guide of the *Telepath* repository page 2. 71
C.6 The query planner guide page 1. 72
C.7 The query planner guide page 2. 73
C.8 The query planner guide page 3. 74
C.9 The query planner guide page 4. 75
C.10 The query planner guide page 5. 76
C.11 The guide on adding an extra physical operator to *Telepath* page 1. 77
C.12 The guide on adding an extra physical operator to *Telepath* page 2. 78
C.13 The guide on adding an extra physical operator to *Telepath* page 3. 79
C.14 The guide on adding an extra physical operator to *Telepath* page 4. 80

List of Tables

2.1 Recursive definition of the *size* of an arbitrary expression $r \in RPQ$. 7

5.1 Mapping from logical operator to physical operators, for use in the physical operator enumerator. 30
5.2 Size of the physical plan space for *path queries*. 36

6.1 Number of nodes and edges in both the LUBM and the Advogato dataset, followed by the size of the result set for each query. . . . 42
6.2 The number of indexed paths and the storage space taken for both the $k = 1$ and the $k = 2$ index. 42
6.3 Cardinality estimation experiment on the LUBM dataset showing the size of each query result set compared to the relation statistics based cardinality estimation. 43
6.4 Cardinality estimation experiment on the Advogato dataset showing the size of each query result set compared to the relation statistics based cardinality estimation. 44
6.5 Combined error of the relation statistics based cardinality estimation experiment for both datasets, grouped by the length of the paths to which the queries evaluate. 44
6.6 Query planning experiment on the LUBM dataset showing the minimum and maximum number of intermediate results for the space of physical plans and how it compares to the chosen physical plan. 45
6.7 Query planning experiment on the Advogato dataset showing the minimum and maximum number of intermediate results for the space of physical plans and how it compares to the chosen physical plan. 46
6.8 Combined error of the query planning experiment for both datasets, grouped by the length of the paths to which the queries evaluate. 46
6.9 Query evaluation experiment on the LUBM dataset showing the timings (ms) to retrieve the last query answer in `Neo4j` and *Telepath* with *k-path* indices available for $k = 1$, and for $k = 2$. 47
6.10 Query evaluation experiment on the Advogato dataset showing the timings (ms) to retrieve the last query answer in `Neo4j` and *Telepath* with *k-path* indices available for $k = 1$, and for $k = 2$. . . 48
6.11 Combined speedup of query evaluation in *Telepath* compared to `Neo4j` for both datasets, grouped by the length of the paths to which the queries evaluate. 48
6.12 Combined speedup of query evaluation in *Telepath* compared to `Neo4j` for both datasets, grouped by the size of the query result set. 48

Listings

2.1 Example simple regular path query. 7
2.2 Example complex regular path query. 7
4.1 Example expression $q_{ex} \in RPQ$ 16
4.2 Schematic logical plan for q_{ex} 16
4.3 Schematic visualization of regular path query processing into a logical plan. 17
4.4 Schematic physical plan for q_{ex} 17
4.5 Schematic physical plan evaluation for q_{ex} 18
4.6 Illustrative example of logical plan flattening. 19
4.7 Illustrative example of the generation of subtrees of size 2, and an example of the generation of subtrees of size 3. Both full and partial subtrees are generated in these examples. 19
4.8 Illustrative example of subtree containment checking for full subtrees. 20
4.9 Illustrative example of subtree containment checking for partial subtrees. 20
4.10 Schematic illustration for physical operator enumeration when joining two physical plans. This example shows physical operator enumeration for the `CONCATENATION` operator. 21
5.1 Example physical plan P_{ex} where each subtree which is rooted by an operator is numbered. 34
6.1 Regular path queries intended for the LUBM dataset. 40
6.2 Regular path queries intended for the Advogato dataset. 41
B.1 Regular path queries intended for the LUBM dataset. 61
B.2 Cypher queries intended for the LUBM dataset. 61
B.3 Regular path queries intended for the Advogato dataset. 62
B.4 Cypher queries intended for the Advogato dataset. 63
C.1 Directory structure of *Telepath* 64
C.2 Interface for physical operators 65

Chapter 1

Introduction

Massive graph-structured data collections are omnipresent in modern data management scenarios such as social networks, linked open data, and chemical compound databases.

There is a need for natural and expressive ways to query efficiently over these massive graphs.

1.1 Motivation

A fundamental paradigm in graph query languages is the so-called regular path query (RPQ). A regular path query specifies a regular expression[1] over the labels of the edges in a graph. The regular expression forms a language to which the sequence of edge labels along a path have to adhere. The query answer of a regular path query consists of the paths such that the word formed by the sequence of its edge labels is in the language recognized by the regular expression.

The selection and manipulation of paths constitute the core of querying graph datasets. However, the feasibility of a path-centric approach to indexing large graphs is an open problem [1, 21]. To date, one study has been performed on the benefits of path indexing for processing graph queries in industry-strength graph databases in [23].

The study of [23] on *path* indexing shows that use of the index yields, on average, orders of magnitude faster query processing times compared with `Neo4j`[2], a popular open-source native graph database which offers features such as being fully transactional and supporting a declarative graph query language, Cypher. However, the work by [23] did not study the full query processing process, i.e., from query processing to query plan generation. Their work focused on studying the underlying *path* index, rather than providing and studying a full query processing framework. A comprehensive query processing framework leveraging a *path* index is presented in this thesis.

[1] A sequence of characters that define a search pattern.
[2] `Neo4j` documentation can be found at `https://neo4j.com/docs/`.

1.2 Goal

We have set out to design, engineer and test a modular path-index based graph database engine with support for evaluation of regular path queries. We name the approach that we develop ***Telepath***.

Telepath follows up on path indexing techniques as proposed in [23] by incorporating such a path index in the full lifecycle of a query. The lifecycle of a query includes query parsing, query planning, and query evaluation among others.

Furthermore, it will be engineered in a generic and modular fashion which enables other researchers to use *Telepath* as a framework and foundation for the engineering and testing of their graph database extensions.

1.3 Contributions

The main contributions of *Telepath* can be summed up into the following four main areas: relation statistics based cardinality estimation, dynamic programming based query planning, query evaluation using a disk-based *k-path* index, and design, engineering, and testing of the proposed approach. The following enumeration expands on the contributions into these areas.

1. Relation statistics based cardinality estimation.
   - Novel implementation of the relation statistics based cardinality estimation ideas as described in [29].
   - Accurate approach for estimating intermediate and resulting query answer sets for *path queries*.
2. Dynamic programming based query planning.
   - Novel implementation of the dynamic programming based query planner ideas as described in [18].
   - Candidate physical plans are generated in a bottom-up fashion, pruning suboptimal physical plans by limiting the search space, while still choosing a near optimal physical plan for evaluation.
   - We show that our dynamic programming based query planner algorithm chooses a physical plan in time polynomial in the size of the given regular path query, while the space of physical plans grows exponentially by the size of the given regular path query.
   - The disk-based *k-path* index is honored and generically incorporated in the cost model, enabling later modifications on the cost model.
3. Query evaluation while leveraging a disk-based *k-path* index.
   - We use an existing implementation of a *k-path* index as described in [23].
   - We provide our own concept and implementation of end-to-end query evaluation while leveraging a disk-based *k-path* index.

- Query evaluation timings appear to be significantly faster compared to a current state of the art graph database engine.

4. Prototype engineering.

    - The implementation has been done with a modular and generic approach to suit extensions and modifications by other researchers.
    - We followed code quality standards which enforce readable, durable and maintainable source code.
    - Implemented features have been unit tested which ensures a correctly working prototype when applying extensions and modifications.
    - The source code of *Telepath* has been made publicly available to contribute to the open-source community. The repository is hosted on `GitHub`[3] at https://github.com/giedomak/Telepath.

[3] `GitHub` is a web-based version control repository hosting service.

Chapter 2

Preliminaries

2.1 Graphs

We adopt a basic model of finite, edge-labeled, directed graphs $G = \langle N, E, \mathcal{L} \rangle$, where: N is a finite set of nodes, $\mathcal{L}$ is a finite set of edge labels, and $E \subseteq N \times \mathcal{L} \times N$ is a set of labeled directed edges. As an example, a graph G_{ex} over the vocabulary $\mathcal{L} = \{$ `associatedWith, enrolledAt, friendOf, parentOf` $\}$ is shown in Figure 2.1.

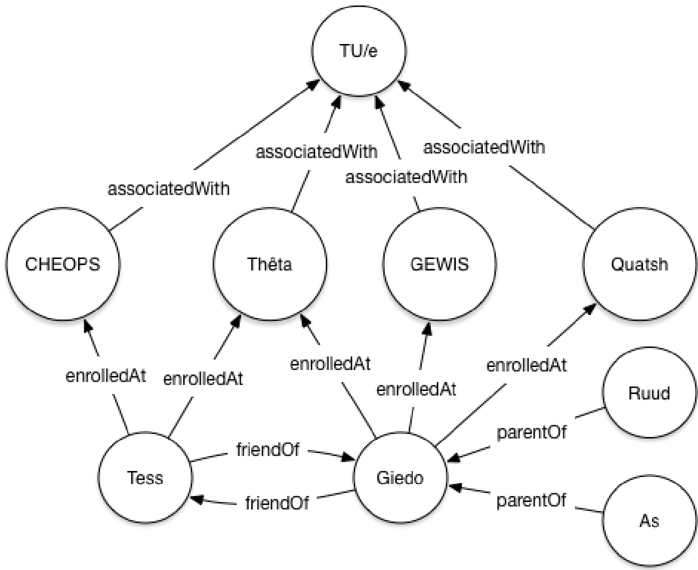


Figure 2.1: A graph G_{ex} over vocabulary $\mathcal{L} = \{$ `associatedWith, enrolledAt, friendOf, parentOf` $\}$.

For each $(s, l, t) \in E$, we say s has an outgoing edge to t having label l, and t has an incoming edge from s having label l.

2.2 Paths

For each edge $(s, l, t) \in E$, we say there is a path of length one from s to t having label l and there is a path of length one from t to s having label l^{-1}.

Let k be a natural number. If $k = 0$, we say there is a *k-path* from s to s, for every $s \in N$. If $k > 0$, for every $s, t \in N$, we say there exists a *k-path*, i.e., a path of length k, from s to t if there exist $n_0, \ldots, n_k \in N$ with edge labels $l_1, \ldots, l_k \in \mathcal{L}$ such that $n_0 = s$, $n_k = t$, and, for $0 < i \leq k$ there is a path of length one from n_{i-1} to n_i having label l_i or having label l_i^{-1}.

2.3 Path queries

Path queries are specified by a regular expression over the sequence of edge labels along a *path*. Such queries, we call regular path queries (RPQ) [2, 4]. The query answer of a regular path query is the projection of the source node and target node of every *path* in the graph such that the sequence of edge labels along the *path* forms a word in the language recognized by the regular expression.

The atomic elements which build a regular path query, are edge labels, binary operators, and unary operators. Parenthesis are used to indicate a particular order of evaluation.

Definition 2.1. The query answer of a regular path query containing exactly one and only one edge label, l, will consist of all node pairs (s, t) such that there is a path of length one from s to t having edge label l.

Definition 2.2. Given edge label $l \in RPQ$, we define $!l$ to denote the inverse edge label of l. The query answer of $!l$ will consist of all node pairs (s, t) such that (t, s) appears in the answer of l.

Definition 2.3. Given expressions $e, f \in RPQ$, we define the `CONCATENATION` binary operator as $e \,/\, f$. The query answer of $e \,/\, f$ consists of every node pair $(s, t) \in N \times N$, such that there exists $x \in N$, where (s, x) is in the answer of e, and (x, t) is in the answer of f.

Definition 2.4. Given expressions $e, f \in RPQ$, we define the `UNION` binary operator as $e \mid f$. The query answer of $e \mid f$ consists of every node pair $(s, t) \in N \times N$, such that (s, t) is in the answer of e, or (s, t) is in the answer of f. The semantics of evaluating $e \mid f \in RPQ$ behave the same as evaluating $e \cup f$.

Definition 2.5. Given expression $e \in RPQ$, we define the `KLEENE STAR` unary operator[1] as e^*. The query answer of e^* consists of every node pair $(s, t) \in N \times N$, such that (s, t) is in the answer of $\epsilon \mid e \mid (e/e) \mid (e/e/e) \mid \ldots$, where ϵ is the empty word.

[1] Kleene star and Kleene plus reference: http://www.oxfordreference.com/view/10.1093/oi/authority.20110803100039649

Definition 2.6. Given expression $e \in RPQ$, we define the KLEENE PLUS unary operator as $e^+ \in RPQ$. The query answer of e^+ consists of every node pair $(s, t) \in N \times N$, such that (s, t) is in the answer of $e \mid (e/e) \mid (e/e/e) \mid \ldots$.

See Appendix A for the complete definition of the regular path query grammar.

Definition 2.7. Let $|r|$ denote the *size* of regular path query r. Depending on the operator, we define $|r|$ recursively as shown in Table 2.1.

	Edge label	Binary operators		Unary operators	
		r_1/r_2	$r_1 \mid r_2$	r_1^*	r_1^+
$\lvert r \rvert$	1	$\lvert r_1 \rvert + \lvert r_2 \rvert$	$\lvert r_1 \rvert + \lvert r_2 \rvert$	$\lvert r_1 \rvert + 1$	$\lvert r_1 \rvert + 1$

Table 2.1: Recursive definition of the *size* of an arbitrary expression $r \in RPQ$.

As an example, the following regular path query selects all node pairs (s, t) such that person s is enrolled to an association which is associated with university t:

```
enrolledAt / associatedWith
```

Listing 2.1: Example simple regular path query.

This example regular path query over graph G_{ex} as shown in Figure 2.1 evaluates to the result set { (Tess, TU/e), (Giedo, TU/e) }.

As a second example, the following regular path query selects all node pairs (s, t) such that university s has an association which has an enrolled person of which person t is either a friend of the enrolled person or the parent of the enrolled person:

```
!associatedWith / !enrolledAt / ( friendOf | !parentOf )
```

Listing 2.2: Example complex regular path query.

This example regular path query over graph G_{ex} as shown in Figure 2.1 evaluates to the result set { (TU/e, Tess), (TU/e, Giedo), (TU/e, Ruud), (TU/e, As) }.

2.4 k-path index

The definition of a *k-path* is stated in Section 2.2. The use of an index over *k-paths* can increase the performance of *path* queries. Leveraging such a *k-path* index during the full lifecycle of a query is one of the main topics of this thesis.

A *k-path* index holds information based on features of a graph in order to identify and describe all paths of the graph. The ordered set of edge labels along a path is used as a pattern to store and retrieve the node pairs which are connected by such a path. Using this edge label pattern, the *k-path* index retrieves the query answer by using its index instead of traversing the graph.

Chapter 3

Related work

3.1 Query optimization

A query can be executed in many different ways by the underlying database engine. The strategy which is used by the database during query evaluation is encoded in a physical plan. The *costs* of such plans, that is, the amount of time that they need to run, can vary by orders of magnitude. This motivates the problem of choosing the physical plan with the lowest possible cost.

Query optimization is a large research area in the database field which attempts to find answers to the problem of choosing an effective physical plan. The area of this problem has been surveyed extensively, for example in [12, 27], of which some findings are presented in the following paragraphs.

Query optimization can be abstracted into a *rewriting* and a *planning* stage. See Figure 3.1 for a graphical overview of the query optimization architecture.

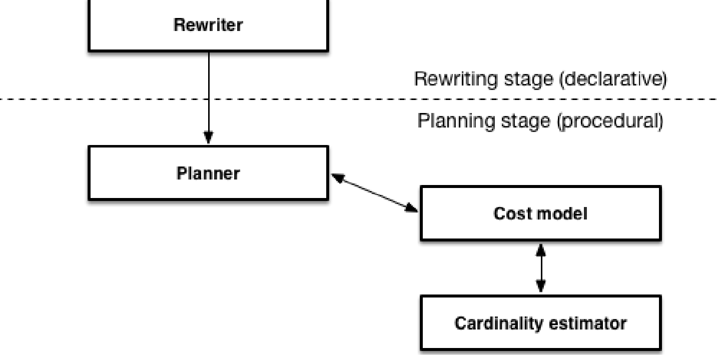


Figure 3.1: Query optimization architecture as described in [12].

The rewriting stage rewrites a query into equivalent queries intended to be more efficient. For example by flattening out nested queries. The rewriting stage does not take the actual database engine, and therefore, the actual query cost, into account. The rewriting stage only relies on static characteristics of the query to create a set of candidate physical plans.

The planning stage is the most crucial stage for any query optimizer. Query planning explores the space of physical plans. It compares these plans based on estimates of their cost, which are derived by the *cost model* and the *cardinality estimator* modules and selects the physical plan with the overall lowest cost to be used to generate the answer to the original query.

Several novel search strategies have been proposed to employ the planning stage to explore the search space of possible physical plans. Such algorithms include simulated annealing, iterative improvement, and two-phase optimization [13]. However, the most important strategy is based on dynamic programming. Dynamic programming constructs physical plans by iterating on the number of relations joined so far, while pruning physical plans known to be suboptimal. For queries with less than ten joins, dynamic programming has proven to be very effective [12].

3.2 Cardinality estimation

As mentioned in Section 3.1, the essence of query optimization is choosing an effective physical plan to be evaluated by the database engine. Cardinality estimation of intermediate and resulting relations is a major component of query optimization since the cost of physical plans can vary by orders of magnitude depending on the cardinalities of these intermediate and resulting relations. Therefore, this motivates the problem of estimating the cardinalities of these relations accurately and efficiently.

Cardinality estimation is an established research topic within the database community. Extensive surveys on the subject of cardinality estimation are found in [17] and [3] for example. The authors of these papers show that there are three primary methods to efficiently estimate cardinalities: relation statistics, histograms, and sampling. Where estimations based on relation statistics are viewed as the simplest method, however, most commercial database management systems base their estimates on histograms.

3.2.1 Histograms

In a histogram of an attribute a of a relation R, the domain of values of a is partitioned into buckets. Within each bucket, a uniform distribution is assumed. For any bucket b in a histogram, if a value $v_i \in b$, then the cardinality f_i of v_i is approximated by $\sum_{v_j \in b} f_j/|b|$. Such histograms assume a uniform distribution over the attribute domain within each bucket and are therefore not very effective in real-world scenarios [22].

There are various other histogram types proposed by researchers which provide better estimations by easing the uniform distribution assumption. For example, in equi-width histograms [15, 20], regardless of the cardinality of each attribute value in the data, the number of consecutive attribute values associated with each bucket is the same. Another class of histograms, equi-depth [15], provides a lower worst-case and a lower average error for some selection queries.

3.2.2 Relation statistics

As described in Section 3.2, cardinality estimation based on relation statistics is one of the three primary methods to estimate cardinalities of intermediate and

resulting relations. In the case of a graph database, statistics over the relations of a given graph describe aspects such as the number of nodes, the number of edge labels, the distribution of edge labels, and the correlation between edge labels and *paths*.

For example, a statistics store which contains for each edge label in the alphabet the number of distinct node pairs which are connected by all paths of length one having the given edge label.

3.3 Path query evaluation

The evaluation of *path queries* using indices has been studied in the context of many kinds of databases, tree and relational databases for example in [5, 7, 19, 28]. Papers such as [8] and [26] have benchmarked different database systems. This section presents some of these findings.

The authors of [26] performed a comparison of the `Neo4j` graph database against the Oracle relational database using the Green-Marl domain specific language [11]. They compared the performance of Dijkstra's shortest path algorithm in a graph database environment with a relational database environment. To model the graph dataset in the relational database, they built an index based on a relation table with columns for the source node, the destination node, and weight. The results of this experiment show that the relational database with an optimized SQL query always performs equally or better than the native graph database `Neo4j`. The authors concede that in certain query types, `Neo4j` outperforms the relational solution. Those solutions tend to be ones with a large number of joins caused by a query which evaluates to a long *path*. In these cases, the number of joins needed cripples the relational database.

A more comprehensive comparison of query evaluations of different databases, and therefore different indexing structures, is presented in [8]. These experiments are conducted on a synthetic dataset with queries which are typically found in the real world. The dataset is the Lehigh University Benchmark, a well-known RDF[1] benchmark containing universities, departments, professors, students, and courses. In their experiment, they compared relational databases with the `Neo4j` and Sparksee graph databases. The results show that even for typical pattern matching operations such as triangle patterns, even unoptimized evaluations in the relational databases perform better than an equivalent evaluation with either of the graph databases in terms of query evaluation time.

The authors of [24] investigated the effectiveness of *path* indexing for accelerating query processing in graph database systems, using `Neo4j` to compare their findings. They present a novel path index design which supports efficient ordered access to paths in a graph dataset, which is fully persistent and designed for external memory storage and retrieval. Their results show that their proposed disk-based *k-path* index yields query execution times from 2x up to 8000x faster than `Neo4j`.

[1] Resource Description Framework, `https://www.w3.org/RDF/`.

Chapter 4

The design and engineering of *Telepath*

This chapter describes the design choices and engineering highlights encountered while creating *Telepath*, our path-index based graph database engine.

4.1 Architecture

While designing the architecture of *Telepath*, one specific goal has to be regarded. The architecture should enable other researchers to use *Telepath* as a building block for their graph database extensions. In other words, *Telepath*'s architecture is as generic and modular as possible.

Figure 4.1 shows a schematic overview of the architecture of *Telepath*. This overview shows the dependencies and connections between modules, as well as a brief description of the responsibilities for each module. Keep in mind that each module can be easily extended with new functionality, or replaced by a new module.

A small description of each module from the schematic overview in Figure 4.1 is stated below.

- **Parser:** Parse the regular path query into a logical plan, our internal representation of a regular path query.
- **Query planner:** Choose an effective physical plan by using heuristics and cost-based enumeration.
- **Evaluation engine:** Evaluate the chosen physical plan to produce a query answer by evaluating each physical operator in the chosen physical plan.
- **Cost model:** Estimate the cost of a physical plan by using cost functions for each physical operator.
- **Cardinality estimation:** Estimate the cardinality of intermediate and resulting query answers for use in the cost functions by using relation statistics of the *k-path* index.
- **PathDB:** The *k-path* index which is used to produce query answers for path queries.
- **Physical library:** Collection of the physical operators which hold information regarding their cost function and their evaluation algorithm.

- **Memory manager:** Disk-based memory buffer for materialising intermediate and resulting query answers.
- **Plan optimization strategies:** Rule-based optimization of logical plans using heuristics.
- **Data models:** Collection of data models used throughout all modules, such as a model for nodes and edges.

4.1.1 Engineering

The directory structure of any engineering project maintains the overview of the separation of modules. Since *Telepath* is designed in a modular and generic fashion, the directory structure mirrors the modular project setup. Appendix C shows the directory structure of *Telepath*.

Another crucial part of creating a generic and modular prototype is the use of interfaces for each module. Interfaces abstract away the public function calls which are allowed by any interacting class. Any class which implements an interface must provide an implementation for all the public functions defined by the interface. Essentially, we define the possible ways of communication and integration with each module in its interface. Different module implementations, which adhere the contract, can be interchanged.

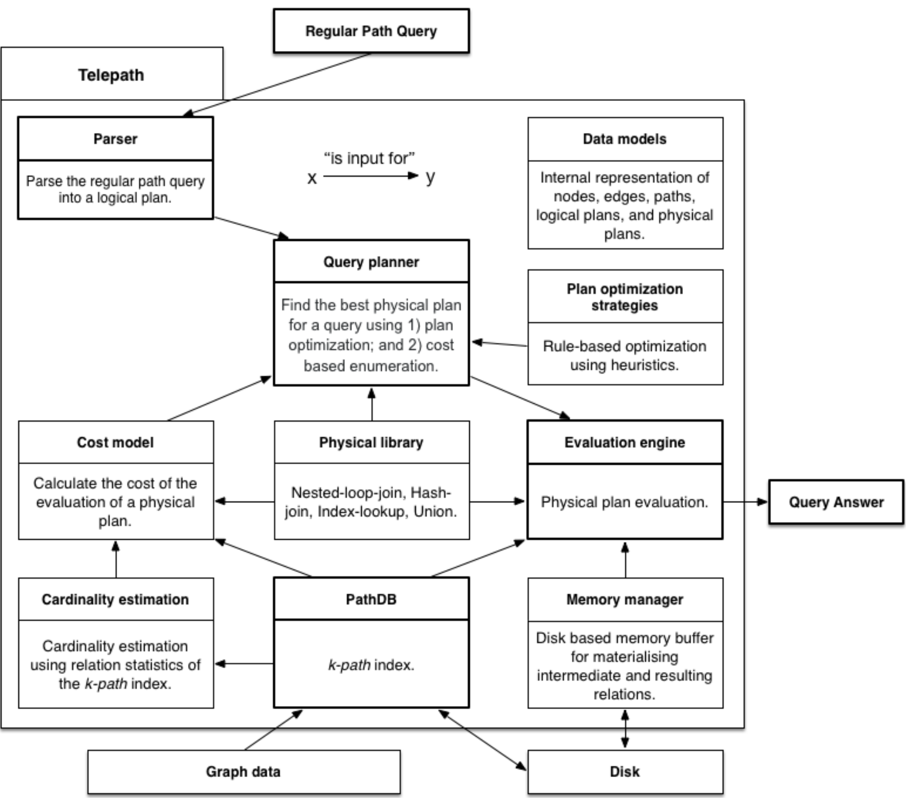


Figure 4.1: Schematic overview of the *Telepath* architecture.

4.2 Life of a Query

The essence of the life of a query within *Telepath* is summarized in the steps as shown in the enumeration below. Each step shows a schematic illustration of the result produced in that step. The goal of this section is to give a general understanding of the life of a query within *Telepath*. References to sections with in-depth details are attached to each step.

4.2.1 Query input

A regular path query is given as input to *Telepath*. The regular path query language is currently the only query language which is supported by *Telepath*. However, any query language can be easily added, by nature of the generic and modular design.

The following regular path query will be used as an example throughout this section.

```
a/(b/c)
```

Listing 4.1: Example expression $q_{ex} \in RPQ$

Where a, b and c are edge labels, and / represents the `CONCATENATION` logical operator. The definition of this query language can be found in Section 2.3.

4.2.2 Parse the input

A given regular path query is parsed into the internal representation of such an expression, and we name it a *logical plan*. Logical plans are represented by a tree data structure. See the following Listing for a schematic example.

```
CONCATENATION
  /       \
 a    CONCATENATION
         /    \
       b      c
```

Listing 4.2: Schematic logical plan for q_{ex}

To interpret regular path queries, *Telepath* uses the `ANTLR`[1] tool, which stands for "ANother Tool for Language Recognition". `ANTLR` is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It is widely used to build languages, tools, and frameworks. From a given grammar, `ANTLR` generates a parser that can build and walk parse trees.

The basic definition of a regular path query can be found in Section 2.3. From this basic definition, we created a grammar which is attached in Appendix

[1]`ANTLR`: http://www.antlr.org/.

```
Regular path query:

    a/(b|(c/d))/e

Logical plan:

            CONCATENATION
              /      \
    CONCATENATION     e
        /   \
      a     UNION
           /   \
          b   CONCATENATION
                 /     \
                c       d
```

Listing 4.3: Schematic visualization of regular path query processing into a logical plan.

A. Using this grammar together with ANTLR, we can parse regular path queries consisting out of edge labels, unary operators, binary operators, and parenthesis. Listing 4.3 shows an extended example of how query input is processed into a logical plan.

4.2.3 Physical plan selection

The query planner generates a set of candidate physical plans and chooses the one of least estimated cost in a bottom-up fashion. See the following Listing for a schematic example of the chosen physical plan which evaluates expression q_{ex}. This example assumes the availability of a large enough k value for the disk-based *k-path* index.

```
          INDEX_LOOKUP
            /  |  \
           a   b   c
```

Listing 4.4: Schematic physical plan for q_{ex}

See Section 4.3 for more details on query planning.

4.2.4 Evaluate the physical plan

The physical plan, as chosen by the query planner, is evaluated in a bottom-up fashion. All intermediate results are materialized through the memory manager, to control any disk-based action.

The physical plan as shown in Listing 4.4 leverages an available disk-based *k-path* index. The design of the disk-based *k-path* index used by *Telepath* is called

`PathDB`[2], a disk-based *k-path* index developed by the authors of [23]. See the following Listing for a schematic example of evaluating q_{ex} using `PathDB`.

```
kPathIndex.search(
        physicalPlan.indexLookupId() // Represents: a/b/c
)
```

Listing 4.5: Schematic physical plan evaluation for q_{ex}

4.3 Query planning

For each logical plan, many corresponding physical plans exist. A physical plan assigns concrete evaluation algorithms to be used for each logical operator specified in a logical plan. The cost of a given physical plan is estimated based on a cost model. This model is carefully tuned based on the concrete evaluation algorithm, execution hardware and cardinalities of participating relations. See Section 4.3.5 for more details on the cost model.

As described in Section 3.1, for queries with less than ten joins, dynamic programming has proven to be very effective. The authors of [18] propose their derivation on the bottom-up approach of dynamic programming and call their algorithm `DPsize`. Our design of the query planner is based on this algorithm. The query planner of *Telepath* chooses the physical plan with the lowest estimated cost in a bottom-up fashion when given a logical plan. Chapter 5 contains an in-depth explanation of our dynamic programming based query planner. In essence, we construct a candidate physical plan of size n, by combining two lowest-cost physical plans of size k and $n - k$.

Illustrative examples of the query planning phase are shown in the following subsections, while all design details are stated in Chapter 5. We will show in Section 5.7 and Section 5.8 that we chose the dynamic programming approach since it enables to create an algorithms that is able to choose a physical plan in polynomial time while the space of physical plans grows exponential by the size of the *path query*.

4.3.1 Flatten the logical plan into a multi-children tree

Logical plans are flattened to prepare them for the subtree generator. Flattened logical plans provide much fewer edge cases which have to be traversed to generate partial subtrees. See Listing 4.6 for an illustrative example of logical plan flattening.

4.3.2 Generate subtrees of a given size

Since the physical plan with the lowest estimated cost of size n is constructed by joining two physical plans of size k and $n{-}k$, a mechanism to retrieve all subtrees

[2] PathDB: A data store for graph paths, http://www.pathdb.com/.

```
Given logical plan:

        CONCATENATION
            /    \
           a    CONCATENATION
                  /    \
                 b      c

Flattened logical plan:

        CONCATENATION
           /  |  \
          a   b   c
```

Listing 4.6: Illustrative example of logical plan flattening.

of a given size has to be provided. These subtrees can either be full subtrees or partial subtrees. Partial subtrees are generated recursively by either using a sliding window which moves over the children or by using all permutations of the children of a given logical plan. This choice is dependent on the operator being ordered or unordered.

The following Listing shows the partial and full subtrees generated by using a sliding window recursively over the children of the given logical plan.

```
Given logical plan:

         CONCATENATION
        /  |     |  |  \
       a  UNION  e  f   g
          / | \
         b  c  d

Subtrees of size 2:

       UNION   UNION   UNION   CONCATENATION    CONCATENATION
        / \     / \     / \       /   \            /   \
       b   c   c   d   b   d     e     f          f     g

Subtrees of size 3:

       UNION      CONCATENATION
       / | \        /   |   \
      b  c  d      e    f    g
```

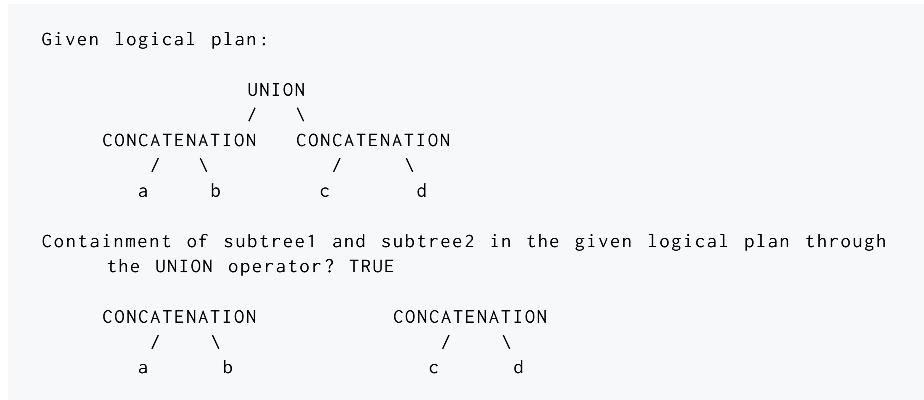
Listing 4.7: Illustrative example of the generation of subtrees of size 2, and an example of the generation of subtrees of size 3. Both full and partial subtrees are generated in these examples.

4.3.3 Check containment of subtrees

The subtrees of size k and $n - k$ will be checked for containment in the logical plan when joined by any applicable logical operator.

When two subtrees are jointly contained in the logical plan when joined by an operator, the query planner has to choose the physical plan with the lowest estimated cost for this newly created subtree.

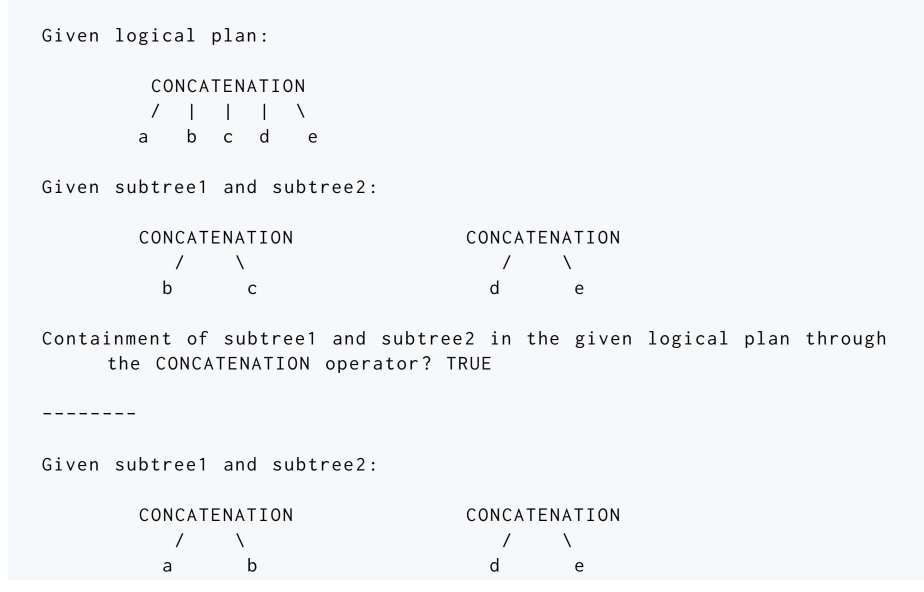
For example, full subtrees `subtree1` and `subtree2` are contained in the given logical plan through the `UNION` operator in the following example.



```
Given logical plan:

                 UNION
                 /   \
     CONCATENATION   CONCATENATION
         /   \          /     \
        a     b        c       d

Containment of subtree1 and subtree2 in the given logical plan through
     the UNION operator? TRUE

     CONCATENATION           CONCATENATION
         /    \                  /    \
        a      b                c      d
```

Listing 4.8: Illustrative example of subtree containment checking for full subtrees.

In the next example, the partial subtrees `subtree1` and `subtree2` are jointly contained in the given logical plan when joined by the `CONCATENATION` operator, only when the common parent does not have a middle child splitting both partial subtrees.



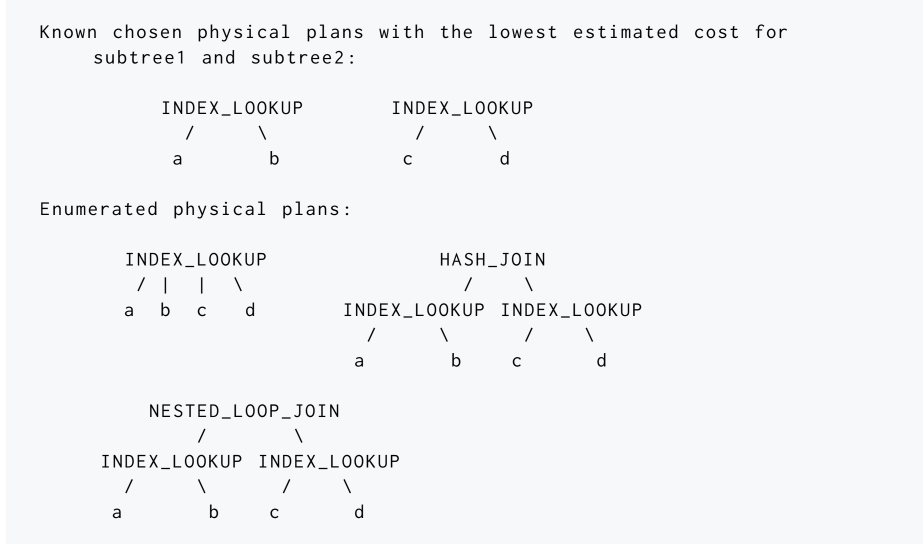
```
Given logical plan:

         CONCATENATION
         /  |  |  |  \
        a   b  c  d   e

Given subtree1 and subtree2:

        CONCATENATION              CONCATENATION
           /    \                     /    \
          b      c                   d      e

Containment of subtree1 and subtree2 in the given logical plan through
     the CONCATENATION operator? TRUE

--------

Given subtree1 and subtree2:

        CONCATENATION              CONCATENATION
           /    \                     /    \
          a      b                   d      e
```

```
Containment of subtree1 and subtree2 in the given logical plan through
     the CONCATENATION operator? FALSE
```

Listing 4.9: Illustrative example of subtree containment checking for partial subtrees.

4.3.4 Enumerate physical operators

As an example, when two subtrees joined by the `CONCATENATION` logical operator are contained in the logical plan, we enumerate the logical operator by which both subtrees are joined, i.e., `CONCATENATION`, into applicable physical operators supported by *Telepath*, e.g., `HASH-JOIN`, `NESTED-LOOP-JOIN`, `INDEX-LOOKUP`. The physical plan with the lowest estimated cost for both subtrees is already chosen since we choose the physical plan with the lowest estimated cost in a bottom-up fashion. See the following Listing for an example of the enumeration of two subtrees which are joined by the `CONCATENATION` logical operator.



```
Known chosen physical plans with the lowest estimated cost for
    subtree1 and subtree2:

         INDEX_LOOKUP       INDEX_LOOKUP
           /     \            /     \
          a       b          c       d

Enumerated physical plans:

       INDEX_LOOKUP              HASH_JOIN
        / |  |  \                   /    \
       a  b  c   d       INDEX_LOOKUP  INDEX_LOOKUP
                           /     \      /    \
                          a       b    c      d

         NESTED_LOOP_JOIN
             /       \
     INDEX_LOOKUP  INDEX_LOOKUP
       /     \      /    \
      a       b    c      d
```

Listing 4.10: Schematic illustration for physical operator enumeration when joining two physical plans. This example shows physical operator enumeration for the `CONCATENATION` operator.

See Section 5.6 for further details on physical operator enumeration.

4.3.5 Costing physical plans

The query planner has to choose the physical plan with the lowest cost from the set of physical plans generated by the physical operator enumerator. Each

available physical operator in *Telepath* has a simple naïve cost function associated with it. A cost function uses the estimated cardinalities of the relations on which the physical operator operates.

As an example, the cost of the hash join physical operator is $2 * (M + N)$, where M is the estimated cardinality of the first relation, and N is the estimated cardinality of the second relation. Using cardinality estimation and the cost functions from the physical operators, the physical plan with the least amount of estimated cost is chosen for evaluation.

See Section 5.6 for further details on cardinality estimation and cost estimation.

4.4 Status of engineering

This section describes the current status of engineering *Telepath*. Since the main goal of this thesis is to deliver an end-to-end solution to evaluate a regular path query efficiently, each module is firstly engineered with a basic implementation before an improved implementation is considered. This section describes the status of each module.

- **Parser:** The parser has been fully implemented and is able to parse all regular path queries including edge labels, inverse edge labels, unary operators and binary operators as described in Section 2.3 into a logical plan.
- **Query planner:** The query planner is able to generate a physical plan for all logical plans the parser module is able to produce excluding logical plans which contain a unary operator. The dynamic programming based algorithm of the query planner is able to process logical plans containing unary operators, but the physical implementation, cardinality estimation, and cost function are not yet implemented for the unary operators.
- **Evaluation engine:** The evaluation engine is fully implemented for all the physical operators supported by our physical library.
- **Cost model:** Currently we implemented a naïve cost model which only accounts for cardinality estimates, disregarding any additional IO costs or implementation specific optimizations.
- **Cardinality estimation:** We implemented two cardinality estimators, a naïve one and a relation statistics based one. The naïve cardinality estimator produces rough estimates for the union and the concatenation binary operators. The relation statistics based cardinality estimator produces effective estimates for *path queries* which only contain the concatenation binary operator.
- **PathDB:** A recent rebuild of the library containing our *k-path* index implemented the index as a disk-based index. This recent development might contain performance flaws since it has not yet been given a dedicated performance study.

- **Physical library:** The physical library currently contains physical operators for the concatenation binary operator (e.g., hash join and nested loop join), the union binary operator (e.g., stream concatenation) and an index lookup. Each physical operator has a cost function associated with it which contributes to the cost model.
- **Memory manager:** The current implementation is able to serialize to disk and deserialize from disk to control allocated and used memory needed for producing intermediate and resulting query answers.
- **Plan optimization strategies:** Logical plans are currently not optimized by using heuristics.
- **Data models:** Data models for graph related data structures are fully implemented. For example, the path data model contains logic on how it can be serialized, which is used for reading and writing to disk.

Chapter 5

Dynamic programming based query planner

As described in Section 3.1, for queries with less than ten joins, dynamic programming has proven to be very effective. We have seen the basic design of the *Telepath* query planner in Section 4.2, and later in 4.3. This chapter provides in-depth details and an analytical analysis of our query planner.

5.1 Introduction

Finding an effective join order in the field of relational database systems remains one significant and complex problem any cost-based query optimizer has to solve. In [22], the authors propose a dynamic programming algorithm to find an effective join order for a given conjunctive query. More precisely, they propose to generate plans in the order of increasing size. The general idea of their algorithm can be used to derive an algorithm which explores the space of bushy trees. Bushy trees do not restrict that either the left or the right child is always a leaf for every subtree in the tree.

The authors of [18] propose their derivation on the bottom-up approach and call their algorithm `DPsize`. This algorithm still forms the core of state of the art commercial query optimizers like the one of DB2 [6] and still is in the context of distributed database management systems the base for further research on join ordering [16]. Figure 5.1 shows the `DPsize` algorithm in pseudocode.

We developed our derivation of `DPsize` which chooses the physical plan with the lowest estimated cost in a bottom-up fashion for a given logical plan. In essence, we construct an effective physical plan of size n, by combining two effective physical plans of size k and $n - k$.

As we will show in Section 5.7 and Section 5.8, we chose the dynamic programming approach since such algorithms are able to choose a physical plan in polynomial time while the space of physical plans grows exponential.

Our dynamic programming based query planning algorithm as implemented in *Telepath*, called `TelPlan`, is shown in Algorithm 1 in pseudocode.

Algorithm 1: `TelPlan`: Dynamic programming based query planning, leveraging a disk-based *k-path* index.

```
   Input  : A logical plan representing regular path query L with size n.
   Output: A physical plan with the lowest estimated cost equivalent to L.
   /* Initialize all subplans of L with size 1 with an index lookup as its
      best physical plan.                                                  */
 1 for all S ∈ L : |S| = 1 do
 2 |  BestPlan({S}) ← IndexLookup(S)
 3 end
   // size of plan, starting with 2.
 4 for all 2 ≤ size ≤ n ascending do
   |  // size of left subplan
 5 |  for all 1 ≤ s1 < size do
   |  |  // size of right subplan
 6 |  |  s2 ← (size − s1)
 7 |  |  for all S1 ∈ L : |S1| = s1 do
 8 |  |  |  for all S2 ∈ L : |S2| = s2 do
   |  |  |  |  // check for containment in L for all binary operators.
   |  |  |  |  // we start by checking for CONCATENATION (/).
 9 |  |  |  |  if (S1/S2) ∈ L then
   |  |  |  |  |  // enumerate applicable physical operators.
10 |  |  |  |  |  physicalPlans ← enumerateOperators(BestPlan(S1),
   |  |  |  |  |    BestPlan(S2), CONCATENATION)
   |  |  |  |  |  // cost each physical plan using the cost model.
11 |  |  |  |  |  currentPlan ← physicalPlans.sortBy(cost).first
   |  |  |  |  |  // save as best plan if the cost is lower.
12 |  |  |  |  |  if cost(currentPlan) < cost(BestPlan({S1/S2})) then
13 |  |  |  |  |  |  BestPlan({S1/S2}) = currentPlan
14 |  |  |  |  |  end
15 |  |  |  |  end
16 |  |  |  |  if (S1|S2) ∈ L then
   |  |  |  |  |  // Analogous to (S1/S2) ∈ L.
17 |  |  |  |  end
18 |  |  |  end
19 |  |  end
20 |  end
   |  // check for containment in L for all unary operators.
21 |  for all S ∈ L : |S| = size do
22 |  |  if S* ∈ L then
   |  |  |  // Analogous to (S1/S2) ∈ L.
23 |  |  end
24 |  |  if S+ ∈ L then
   |  |  |  // Analogous to (S1/S2) ∈ L.
25 |  |  end
26 |  end
27 end
28 return BestPlan({L})
```

```
DPsize
Input: a connected query graph with relations R =
{R_0, ..., R_{n-1}}
Output: an optimal bushy join tree without cross products

for all R_i ∈ R {
    BestPlan({R_i}) = R_i;
}
for all 1 < s ≤ n ascending // size of plan
for all 1 ≤ s_1 < s { // size of left subplan
    s_2 = s − s_1; // size of right subplan
    for all S_1 ⊂ R : |S_1| = s_1
            S_2 ⊂ R : |S_2| = s_2 {
        ++InnerCounter;
        if (∅ ≠ S_1 ∩ S_2) continue;
        if not (S_1 connected to S_2) continue;
        ++CsgCmpPairCounter;
        p_1=BestPlan(S_1);
        p_2=BestPlan(S_2);
        CurrPlan = CreateJoinTree(p_1, p_2);
        if (cost(BestPlan(S_1 ∪ S_2)) > cost(CurrPlan)) {
            BestPlan(S_1 ∪ S_2) = CurrPlan;
        }
    }
}
OnoLohmanCounter = CsgCmpPairCounter / 2;
return BestPlan({R_0, ..., R_{n-1}});
```

Figure 5.1: Dynamic programming algorithm `DPsize` as described in [18].

5.2 Subproblems

As can be seen in Algorithm 1, five essential subproblems emerge for a given logical plan representing regular path query L. We state these subproblems below.

- P1 Given a size s, find all subexpressions in L of size s.
- P2 Given two expressions $S_1, S_2 \in L$, check if $S_1 \otimes S_2 \in L$ for every binary operator $\otimes$ we support.
- P3 Given expression $S \in L$, check if $S\diamond \in L$ for every unary operator $\diamond$ we support.
- P4 Given two expressions $S_1, S_2 \in L$, and a binary operator $\otimes$ for which $S_1 \otimes S_2 \in L$ holds, join their chosen physical plan by all applicable physical operators for binary operator $\otimes$.
- P5 Given a collection of physical plans, choose the physical plan with the least amount of cost.

The following sections describe the design and engineering aspects for these subproblems.

5.3 `P1`: Subexpressions of a given size

Since the physical plan with the lowest estimated cost of size n is constructed by joining two physical plans with each having the lowest estimated cost of size k and $n - k$, a mechanism to retrieve all subtrees of a given size has to be provided. These subtrees can either be full subtrees or partial subtrees. Partial subtrees are generated recursively by using a sliding window which moves over the children of a given logical plan.

The Algorithm for partial and full subtree generation by using a sliding window recursively over the children of a given logical plan is shown in Algorithm 2.

Algorithm 2: `SubtreesOfSize`: Get all subtrees of a given size.

```
   Input  : (S, n), where S is the logical plan, n is the target size.
   Output: A collection of subtrees from S of size n.
   // Break recursion if we are size n.
 1 if S.size = n then
 2 |  return S
 3 end
 4 LogicalPlans subtrees
   // Iterate over each child.
 5 for child ∈ S.children do
 6 |  accumulatedSize ← child.size
   |  /* If the size of this child is smaller than n, we'll try to
   |     concatenate with our brothers and sisters. We'll traverse
   |     increasingly linearly.                                        */
 7 |  if accumulatedSize < n then
   |  |  /* Trying to find a subList of our children which together have
   |  |     size n.                                                    */
 8 |  |  for child.index < i < S.children.size do
 9 |  |  |  accumulatedSize ← accumulatedSize + S.children[i].size
   |  |  |  /* If we've jumped over size n, we'll just try again with the
   |  |  |     next starting child.                                    */
10 |  |  |  if accumulatedSize > n then
11 |  |  |  |  break
12 |  |  |  end
   |  |  |  /* We've found a subList which has our beloved size n.      */
13 |  |  |  if accumulatedSize = n then
14 |  |  |  |  clone ← S
15 |  |  |  |  clone.children = clone.children.subList(child.index, i)
16 |  |  |  |  subtrees.add(clone)
17 |  |  |  end
18 |  |  end
19 |  else
   |  |  /* So while searching for a subList, our starting child already
   |  |     exceeded size n. Try to find recursively some matches in that
   |  |     subtree.                                                   */
20 |  |  subtrees.add(SubtreesOfSize(child, n))
21 |  end
22 end
23 return subtrees
```

5.4 `P2, P3`: Check containment of expression

The Algorithm for containment checking of two subtrees in a given logical plan is shown in Algorithm 3.

Algorithm 3: `ContainmentCheck`: Check if two given subtrees are contained in a given logical plan through a given operator.

Input : $(S, s1, s2, operator)$, where S is the root logical plan, $s1$ and $s2$ are the two subtrees which should be jointly contained through *operator*.

Output: A boolean value indicating containment.

```
1  LogicalPlans subtrees
   /* If we are dealing with a subtree that is rooted with operator, we only
      have to consider its children since we have flattened logical plans. */
2  for subtree ∈ [s1, s2] do
3  |  if subtree.operator = operator then
4  |  |  subtrees.add(subtree.children)
5  |  else
6  |  |  subtrees.add(subtree)
7  |  end
8  end
   /* Check for each subtree which is operator, if our subtrees list is
      directly contained in its children and its indices are next to each
      other.                                                              */
9  for tree ∈ S.treewalk() do
10 |  if tree.operator = operator then
11 |  |  if containsSublistOfChildren(tree.children, subtreess) then
12 |  |  |  return true
13 |  |  end
14 |  end
15 end
16 return false
```

5.5 `P4`: Physical operator enumeration

When two subtrees are jointly contained in the logical plan, we enumerate the logical operator by which both subtrees are joined using applicable physical operators supported by *Telepath*. Since *Telepath* is built in a generic and modular way, physical operators can be easily added as extensions. Currently, we support the physical operators stated below.

- Hash join

  The hash join algorithm first builds a hash table on the join key of the first relation, it then scans the second relation and probes the hash table to look for matches on the join key. When this is the case, the concatenation of both elements is added to the result set.

  The time complexity of the hash join algorithm as implemented in *Telepath* is $2*(M+N)$, where M and N are the cardinalities of the relations on which

the operator performs.

- Nested loop join

  The nested loop join algorithm scans the first relation, for each element it scans, it scans the second relation to look for matches. When a match is found, the concatenation of both elements is added to the result set.

  The time complexity of the nested loop join algorithm as implemented in *Telepath* is $M * N$, where M and N are the cardinalities of the relations on which the operator performs.

- Index lookup

  Since *Telepath* can leverage a disk-based *k-path* index, this physical operator is included in the physical library.

- Union

  The current implementation of the `UNION` operator in *Telepath* makes use of the `Stream`[1] package. This package supports distinct stream concatenation which is used to merge two streams of results while keeping state to ensure distinct elements in the result set.

Table 5.1 shows the mapping from a logical operator to its applicable physical operators. The physical operator enumerator uses this mapping to generate a collection of physical plans of which the one with the lowest estimated cost will be chosen for evalution.

Logical operator	Physical operator
Concatenation	Hash join
Concatenation	Nested loop join
Concatenation	Index lookup
Union	Distinct stream concatenation

Table 5.1: Mapping from logical operator to physical operators, for use in the physical operator enumerator.

Line 10 of `TelPlan` as seen in Algorithm 1 describes the phase where physical operator enumeration is performed. We can expand this phase of the algorithm with more details as shown in Algorithm 4.

In Algorithm 4 we denote two important elements, *operatorMapping*, and *joinPlans*. The *operatorMapping* is derived from Table 5.1, where each logical operator is mapped into one or multiple physical operators. Regarding *joinPlans*, see Section 4.3.4 for a schematic illustration of this operation.

[1] The `Stream` package summary can be found at https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html.

Algorithm 4: `EnumeratePhysicalPlans`: Apply operator mapping in order to produce physical plans.

Input : $(S_1, S_2, operator)$, where S_1 and S_2 are logical plans representing subexpressions of the global query, and *operator* denotes the logical operator which is being mapped into physical operators.

Output: A collection of physical plans.

```
1  PhysicalPlans plans
   // Retrieve the local optimal physical plans for S1 and S2.
2  P1 ← BestPlan(S1)
3  P2 ← BestPlan(S2)
   // Iterate over each mapping, and produce a physical plan.
4  for rule ∈ operatorMapping(operator) do
5  |  plans ← joinPlans(P1, P2, rule)
6  end
7  return plans
```

5.6 P5: Cost estimation

The query planner has to choose the physical plan with the lowest amount of cost for every physical plan generated by the physical operator enumerator. This physical plan with the lowest estimated cost for each subproblem will be memoized as a local solution, contributing to the global physical plan. To efficiently estimate the cost of a physical plan, cardinality estimation of intermediate results contributes significantly to an accurate cost estimate.

Line 11 of `TelPlan` as seen in Algorithm 1 describes the phase where physical plans are costed using a naïve cost model. We can expand this phase of the algorithm with more details as shown in Algorithm 5. This Algorithm estimates a cost for all physical plans it receives as input and returns the physical plan with the least amount of cost.

Algorithm 5: `TelCost`: Cost a collection of physical plans.

Input : *Plans*, a collection of physical plans.

Output: The physical plan with the least amount of cost.

```
1  BestPlan bp
   /* Memoize the cardinalities and the cost for each physical plan
      recursively.                                                  */
2  for p ∈ Plans do
3  |  p1 ← p.firstRelation
4  |  p2 ← p.secondRelation
5  |  p.cardinality ← estimateCardinalityRecursively(p1, p2, p.operator)
6  |  p.cost ← estimateCostRecursively(p1, p2, p.operator)
   |  // Memoize the physical plan with the least amount of cost.
7  |  if p.cost < bp.cost then
8  |  |  bp ← p
9  |  end
10 end
11 return bp
```

5.6.1 Cardinality estimation

Regarding cardinality estimation, we have chosen to incorporate a relation statistics based cardinality estimator in *Telepath*. This design decision was made since estimations based on relation statistics are viewed as the most straightforward method. Note that *Telepath* can be easily extended with modules for histogram or sampling based cardinality estimation.

The author of [29] describes a method to efficiently estimate cardinalities of *path queries* which might have a length greater than two. They achieve this by combining the statistics maintained over all *paths* of length one, and of length two. They name this collection of statistics the SYNOPSIS, where they refer to the statistics for all *paths* of length 1 as SYN1, and to those of all *paths* of length 2 as SYN2.

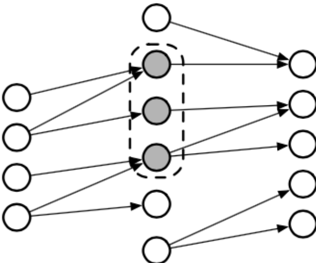


Figure 5.2: Schematic illustration of a join set as presented in [29].

Most database engines which maintain relation statistics assume uniformity, independence, and inclusion. Uniformity holds when all nodes in a join set, as shown in within the dotted line in Figure 5.2, have the same number of tuples associated with them in both the left and right relation. Independence holds when predicates on attributes are independent. Inclusion holds when the domain of join keys overlap in a way such that all keys from the smaller domain have matches in the larger domain. The SYNOPSIS-based cardinality estimation is able to lose the inclusion assumption.

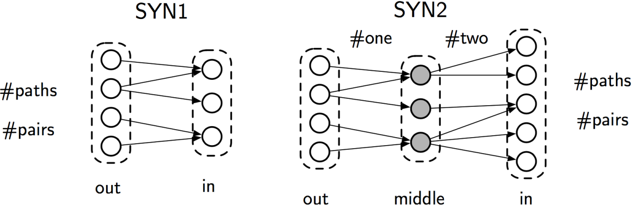


Figure 5.3: Schematic overview of the relation statistics maintained by the SYNOPSIS as presented in [29].

A schematic illustration on how the SYNOPSIS is constructed can be seen in Figure 5.3. SYN1 holds the following statistics for each edge label l: the number

of nodes which have an outgoing edge labeled with l, the number of nodes which have an incoming edge labeled with l, the number of *paths* of length one having its edge labeled with l, and the number of distinct node pairs connected through *paths* of length one having its edge labeled with l.

`SYN2` is constructed similarly as `SYN1`, but holding extra statistics regarding the middle node along a *path* of length two. Instead of collecting statistics for each edge label, `SYN2` collects statistics for each pair of edge labels, l_1, l_2, for which their concatenation, l_1/l_2, is contained in the given graph. In addition to `SYN1`, `SYN2` also collects the following statistics:

- *middle*: the number of nodes which have an incoming edge labeled with l_1 and an outgoing edge labeled with l_2.
- *one*: the number of edges which are labeled with l_1 and originate from one of the nodes which have an outgoing path labeled with l_1/l_2 and go to one of the nodes in *middle*.
- *two*: the number of edges which are labeled with l_2 and originate from one of the nodes in *middle* and go to one of the nodes which have an incoming path labeled with l_1/l_2.

Let T_{r/l_1} denote the query answer of the regular path query r/l_1, where r is an arbitrary regular path query, and l_1 is an edge label. By using `SYN1` and `SYN2`, we can estimate the cardinality of $T_{r/l_1/l_2}$, where l_2 is also an edge label. We can estimate this cardinality using the formula as seen in Figure 5.4.

$$|T_{r/l_1/l_2}| = |T_{r/l_1}| \cdot \frac{l_1/l_2.\#\text{two}}{l_1.\text{in}}$$

Figure 5.4: Formula to estimate the cardinality of *path queries* as seen in [29].

In the formula as stated in Figure 5.4, l_1.in is extracted from `SYN1` which denotes the number of nodes which have an incoming edge labeled with l_1. l_1/l_2.#two is extracted from `SYN2` which denotes the number of edges which are labeled with l_2 and originate from one of the nodes in l_1/l_2.middle and go to one of the nodes which have an incoming path labeled with l_1/l_2.

5.6.2 Cost model

Each available physical operator in *Telepath* has a simple naïve cost function associated with it. The collection of cost functions is named the cost model. Our cost model is naïve since it only accounts for cardinality estimates, disregarding any additional IO costs or implementation specific optimizations.

A cost function uses the estimated cardinalities of the relations on which the physical operator operates. As an example, the cost of the hash join physical operator is $2 * (M + N)$, where M is the estimated cardinality of the first relation, and N is the estimated cardinality of the second relation.

The cost of a physical plan is recursively calculated by utilizing a post-order tree traversal. If we take Listing 5.1 for example, the cost estimation of physical

plan P_{ex} is calculated by adding the estimated cost of the subtree rooted at [2], the estimated cost of the subtree rooted at [3], and applying the cost function $2 * (M + N)$ to the subtree rooted at [1]. Note that the cost function uses the cardinality estimates of the subtree rooted at [2] and the subtree rooted at [3].

Both the cardinality estimates and the cost estimates are memoized for each subtree which is rooted by a physical operator. The cardinality estimates are calculated in the same way as the cost estimate, i.e., by utilizing a post-order tree traversal.

```
Physical plan:

                          [1]HASH-JOIN
                          /          \
            [2]HASH-JOIN          [3]INDEX_LOOKUP
                /      \                   |
     [4]INDEX_LOOKUP  [5]INDEX_LOOKUP      e
           /     \          /     \
          a       b        c       d
```

Listing 5.1: Example physical plan P_{ex} where each subtree which is rooted by an operator is numbered.

5.7 Physical plan selection

The global problem that `TelPlan` has to solve is to find an effective physical plan for a given logical plan. There exist many ways to split this problem into subproblems, since the regular path query represented by the logical plan can be constructed from its subexpressions in many ways.

In the pseudocode of `TelPlan`, we see that all possible splits of the problem into subproblems are considered. I.e., all possible splits of L into s_1 and s_2 are considered by iterating over all sizes $|s_1|$ and $|s_2|$ such that $|s_1| + |s_2| = |L|$. From this, we can conclude that by considering all local solutions for splits of L into s_1 and s_2, we find the global solution for L, in general.

The running time of `TelPlan` can be derived by analyzing the algorithm and its subroutines. Routine `TelCost` selects the physical plan with the lowest estimated cost in $O(|P|)$ time, where $|P|$ is the number of physical plans given to it. Every operation in `TelCost` takes at most $O(1)$ time due to the bottom-up aproach which ensures that the cardinality estimates and the estimated cost of the subproblems of each physical plan are memoized. Set $|P|$ is generated by the `EnumeratePhysicalPlans` routine which takes $O(1)$ time since there are at most three `operatorMappings` to consider. Hence, `TelCost` runs in $O(1)$ time.

To analyze the total running time of `TelPlan`, we look at how many times loops are executed. The loop on lines $[1-3]$ is executed $|L|$ times for all subexpressions in L of size 1. The operation in the loop on lines $[1-3]$ takes at most $O(1)$ time, hence the loop on lines $[1-3]$ runs in $O(|L|)$ time. We then consider the loop on lines $[4-27]$. It contains two inner loops, $[5-20]$ and $21-26$. As described in Section 4.4, the current implementation of `TelPlan` does not include unary

operators, so we restrict our analysis to loop $[5-20]$, which processes the binary operators.

Loop $[7-19]$ executes for every pair of subexpressions S_1, S_2 such that $|S_1| + |S_2| = size$. The two cases in loop $[7-19]$ combine these subexpressions by concatenation or union. Checking if the condition for these two cases holds takes $O(|L|)$ time. For a subexpression of size $size$, these combinations can consist at most of $size-1$ splits into S_1 and S_2. Generating all subexpression of a given size takes $O(|L|)$ time. Since checking the condition takes $O(|L|)$ time and each operation for the pair of subexpressions takes $O(1)$ time, as shown above, loop $[7-19]$ takes $O(|L|^2)$ time.

There are at most $|L| - size + 1$ subexpressions of length $size$ in L for which loop $[4-27]$ executes. Therefore, loop $[4-27]$ runs in:

$$\sum_{size=2}^{|L|} (|L| - size + 1)(size - 1) \cdot O(|L|^2) = \frac{|L|^3 - 3}{6} \cdot O(|L|^2) = O(|L|^5) \tag{5.1}$$

Since loop $[1-3]$ runs in $O(|L|)$ time and loop $[4-27]$ runs in $O(|L|^5)$ time, we have shown that `TelPlan` runs in $O(|L|^5)$ time. This is polynomial in the size of the given regular path query L.

5.8 Size of the physical plan space

This section will analyze the size of the physical plan space for *path queries* which are constructed only out of either edge labels or the `CONCATENATION` binary operator.

For the following analysis, we will consider regular path query expression r, which denotes a *path query* of length n.

$$r = l_1/l_2/.../l_n$$

To analyze the size of the search space for given expression r, we want to know how many physical plans can evaluate expression r. Each one of the concatenations in r can be represented as the concatenation of two subexpressions r_1 and r_2. As we have seen in Section 5.5, joining two physical plans by the `CONCATENATION` operator will result in 3 physical plans, when we assume a large enough k value for the disk-based *k-path* index. Given split r_1/r_2, we obtain the following formula to indicate the number of physical plans S.

$$S_{r_1/r_2} = 3 \cdot (S_{r_1} \cdot S_{r_2})$$

From the formula as stated above, we obtain a recurrence formula for the number of physical plans, S_n, where n is the length of a *path query*.

$$\begin{cases} S_0 = 0 \\ S_1 = 1 \\ S_n = 3 \cdot \sum_{k=1}^{n-1} S_k \cdot S_{n-k} & \text{for } n \geq 2 \end{cases} \tag{5.2}$$

From the recurrence formula as stated above, we compute the first values of S_n using the base case which is defined by $S_1 = 1$. As shown in Table 5.2, the number of physical plans grows quickly with the length of a given *path query*.

n	S_n
1	1
2	3
3	18
4	135
5	1,134
6	10,206
7	96,228
8	938,223
9	9,382,230
10	95,698,746

Table 5.2: Size of the physical plan space for *path queries*.

The recurrence formula from Equation 5.2 relates to the Catalan numbers [10], for which we can use a generating function to obtain a closed form of the recurrence in 5.2 and derive the asymptotic complexity.

We start with the recursively defined equation for $n \geq 2$

$$S_n = 3 \cdot \sum_{k=1}^{n-1} S_k \cdot S_{n-k}$$

Multiply both sides with x^n

$$S_n \cdot x^n = 3 \cdot x^n \cdot \sum_{k=1}^{n-1} S_k \cdot S_{n-k}$$

Since $S_0 = 0$, we can increase the range of our sum to $[0, n]$ and apply a sum to both sides of the equation

$$\sum_{n=2}^{\infty} S_n \cdot x^n = 3 \cdot \sum_{n=2}^{\infty} x^n \cdot (\sum_{k=0}^{n} S_k \cdot S_{n-k})$$

We define $h(x)$ to be

$$h(x) = \sum_{n=0}^{\infty} h_n \cdot x^n$$

Then we can say

$$h(x) = x + (h(x))^2$$

And we find the following solutions for $h(x)$

$$h(x) = \frac{1 \pm \sqrt{1 - 12x}}{6}$$

Since $S_0 = 0$, we find the solution for S_n where $n \geq 1$ to be

$$S_n = -\frac{1}{6}(-12)^n \binom{\frac{1}{2}}{n}$$

If we take the limit of $n \to \infty$, we see that S_n grows asymptotically as

$$S_n \sim \frac{12^n}{n^{3/2}\sqrt{\pi}}$$

Which gives us an upper bound of

$$S_n = O(c^n) \tag{5.3}$$

for some constant c.

We have shown that S_n grows exponentially by the length of the *path query*.

Chapter 6

Experiments and benchmarking

6.1 Experiment goals

This section describes a set of experiments which investigate the value and performance of the main contributions of the implementation of *Telepath*. These main contributions can be summed up to the dynamic programming approach to query planning, the cardinality estimation using relation statistics used for costing physical plans, and the performance gain of query evaluation by using path indices. We define three questions of which we wish to examine and analyze the answers using experiments:

- `G1` How accurate is the relation statistics based cardinality estimator regarding *path queries*?
- `G2` How effective is the dynamic programming based query planner in choosing physical plans?
- `G3` How does the running time of query evaluation in *Telepath*, leveraging a disk-based *k-path* index, compare to the current state of the art graph database engines?

To answer the above questions conclusively, we have designed the experiments which are stated below. These experiments are run on two different datasets since it may be the case that certain properties of these datasets and their varying sizes expose differences in effectiveness.

- `Exp1` We run the relation statistics based cardinality estimator for queries on a variety of datasets. We compare the size of the result set with the estimated cardinality for each query. Results are also grouped by the length of the paths to which the queries evaluate.
- `Exp2` We have designed an experiment where the whole space of physical plans is explored to accumulate their number of intermediate results. The number of intermediate results is used to analyze how often the dynamic programming based query planner chooses the optimal physical plan, and when this is not the case, how significant the error is compared to the optimal physical plan.
- `Exp3` We compare query execution time using our underlying disk-based *k-path* index with the graph database engine `Neo4j` on the queries for the LUBM and the Advogato datasets. We analyze the execution time with *Telepath* having a *1-path* index available, and a *2-path* index available.

The results of the formulated experiments can be found in Section 6.3, while the setup of the experiments, including the environment and the datasets, can be found in Section 6.2.

6.2 Experiment setup

All experiments were performed on a 2.0 GHz i7 processor with 8 GB of main memory and a solid-state drive, running OS X 10.12. Experiments were run on the Lehigh University Benchmark (LUBM) dataset [9], a well-known and widely used synthetic dataset, and the Advogato network dataset [14], a real-world dataset.

We examine *path queries* which are constructed only out of either edge labels or the `CONCATENATION` binary operator. We have chosen to restrict our experiments to queries containing the `CONCATENATION` operator, since our relation statistics based cardinality estimation module is focussed on such queries. As described in Section 1.2, the goal of this project has been to develop an end-to-end solution for query processing. Since the focus during this project has been to develop minimum implementations for each module in the full lifecycle of a query, our cardinality estimation module is currently only focussing on queries containing the `CONCATENATION` operator.

6.2.1 LUBM dataset

The LUBM synthetic dataset is produced by a data generator and can be of arbitrary size. Graphs generated by the LUBM data generator model a university scenario, e.g., nodes represent universities, departments, students, teachers, and courses.

For our experiments, we generated a graph with ten universities, containing approximately 636 thousand unique edges, and 207 thousand unique nodes. We followed the data preparation steps as taken in [8] and [23] as close as possible. Our dataset was not enriched with inferred facts derived from ontology rules. For example, nodes of type “Associate Professor” do not also get the more general label “Professor”. LUBM is provided with 14 different queries whose answer contains *paths* up to a length of 3. Here, we focus on the queries provided by LUBM whose answer contains *paths* of length 3, extended by self-formulated queries whose answer contains *paths* of length 4 and 5. See Listing 6.1 for the regular path queries we formulated to run against the LUBM dataset.

```
// ----- length = 3 -----

Q1: undergraduateDegreeFrom / !subOrganizationOf / !memberOf

Q2: advisor / teacherOf / !takesCourse

Q3: !headOf / worksFor / !subOrganizationOf

Q4: !headOf / worksFor / subOrganizationOf
```

```
// ----- length = 4 -----

Q5: advisor / teacherOf / !takesCourse / advisor

Q6: undergraduateDegreeFrom / !subOrganizationOf / !memberOf / advisor

Q7: !worksFor / teacherOf / !takesCourse / advisor

// ----- length = 5 -----

Q8: !teacherOf / undergraduateDegreeFrom / !subOrganizationOf / !
    memberOf / advisor

Q9: advisor / teacherOf / !takesCourse / advisor / teacherOf
```

Listing 6.1: Regular path queries intended for the LUBM dataset.

6.2.2 Advogato dataset

The Advogato dataset [14] is a trust network from the Advogato online community discussion board for developers of free software. Advogato uses a trust metric to determine a single global trust value for each user. This value is computed by the rating users give to each other. These ratings can be three possible values: apprentice, journeyer, and master. Advogato uses this trust to allow users to access certain administrative controls for the message board.

The graph constructed from this dataset contains nodes which represent the users, and edges which represent the ratings given between users. It contains approximately 56 thousand unique edges, and seven thousand unique nodes. The Advogato dataset does not have specified queries. Therefore we formulate queries whose answer contains *paths* of length three or more, which compares to the LUBM queries. See Listing 6.2 for the regular path queries we formulated to run against the Advogato dataset.

```
// ----- length = 3 -----

Q1: apprentice / apprentice / apprentice

Q2: journeyer / journeyer / journeyer

Q3: master / master / master

Q4: apprentice / journeyer / master

// ----- length = 4 -----

Q5: apprentice / apprentice / apprentice / !journeyer

Q6: apprentice / journeyer / !apprentice / master

Q7: master / apprentice / !master / journeyer

// ----- length = 5 -----
```

```
Q8: apprentice / apprentice / apprentice / apprentice / apprentice

Q9: apprentice / journeyer / !master / !apprentice / master
```

Listing 6.2: Regular path queries intended for the Advogato dataset.

6.2.3 Dataset size

The total number of nodes and edges in both the LUBM and the Advogato dataset, followed by the size of the result set for each query we formulated for the dataset is shown in Table 6.1.

	LUBM	Advogato
Number of nodes	207,456	7,419
Number of edges	636,468	56,461
Q1	2,954,166	458,007
Q2	1,586,587	6,776,541
Q3	2,827	2,627,106
Q4	189	1,008,665
Q5	540,310	3,356,127
Q6	1,165,204	5,932,724
Q7	93,545	6,505,013
Q8	672,697	26,573,816
Q9	1,610,932	59,397,710

Table 6.1: Number of nodes and edges in both the LUBM and the Advogato dataset, followed by the size of the result set for each query.

The number of indexed entries and the storage space taken for both the $k = 1$ and the $k = 2$ disk-based index is shown in Table 6.2.

k-path index	**LUBM**		**Advogato**	
	Entries	Storage	Entries	Storage
$k = 1$	1,272,936	0.14 GB	112,922	0.01 GB
$k = 2$	17,503,776	2.32 GB	6,959,386	0.94 GB
Total	**18,776,712**	**2.46 GB**	**7,072,308**	**0.95 GB**

Table 6.2: The number of indexed paths and the storage space taken for both the $k = 1$ and the $k = 2$ index.

6.3 Experiment results

6.3.1 `Exp1`: Cardinality estimation evaluation

As described in Section 6.1, the goal of this experiment is to answer question `G1`: "How accurate is the relation statistics based cardinality estimator regarding *path queries*?".

To answer this question, we run the relation statistics based cardinality estimator on the queries for the LUBM and the Advogato datasets. We compare the size of the result set with the estimated cardinality for each query using the formula as stated in Figure 6.1.

$$error(r) = \begin{cases} 0, & \text{if } estimate(r) = cardinality(r) \\ \dfrac{estimate(r) - cardinality(r)}{max(estimate(r), cardinality(r))}, & \text{else} \end{cases}$$

Figure 6.1: Formula to measure the quality of a cardinality estimation within the balanced range $[-1, 1]$ as seen in [25].

The formula as stated in Figure 6.1 returns an error rate in the range $[-1, 1]$ which indicates the quality of a cardinality estimation. In this formula r is a given regular path query expression, $estimate(r)$ is the cardinality estimation for expression r, $cardinality(r)$ indicates the cardinality of expression r. We use the absolute value, i.e., $|error(r)|$, when investigating combinations of error rates.

The result of this experiment for the LUBM dataset is shown in Table 6.3, and the result of this experiment for the Advogato dataset is shown in Table 6.4.

LUBM	Result set	Telepath estimation	Error
Q1	2,954,166	185,101	-0.94
Q2	1,586,587	1,571,320	-0.01
Q3	2,827	2,827	0
Q4	189	189	0
Q5	540,310	619,510	+0.13
Q6	1,165,204	72,978	-0.94
Q7	93,545	108,303	+0.14
Q8	672,697	50,360	-0.93
Q9	1,610,932	1,851,899	+0.13
Mean error			**0.36**

Table 6.3: Cardinality estimation experiment on the LUBM dataset showing the size of each query result set compared to the relation statistics based cardinality estimation.

The results of this experiment on the LUBM dataset and the Advogato dataset are combined and grouped by the length of the paths to which the queries evaluate. These combined and grouped results are shown in Table 6.5.

Advogato	Result set	Telepath estimation	Error
Q1	458,007	160,271	-0.65
Q2	6,776,541	2,345,440	-0.65
Q3	2,627,106	1,132,356	-0.57
Q4	1,008,665	427,662	-0.58
Q5	3,356,127	565,589	-0.83
Q6	5,932,724	741,623	-0.87
Q7	6,505,013	1,233,673	-0.81
Q8	26,573,816	971,176	-0.96
Q9	59,397,710	3,482,992	-0.94
Mean error			**0.76**

Table 6.4: Cardinality estimation experiment on the Advogato dataset showing the size of each query result set compared to the relation statistics based cardinality estimation.

Path length	Error
3	0.43
4	0.62
5	0.74
Mean error	**0.56**

Table 6.5: Combined error of the relation statistics based cardinality estimation experiment for both datasets, grouped by the length of the paths to which the queries evaluate.

6.3.2 Exp2: Query planning evaluation

As described in Section 6.1, the goal of this experiment is to answer question G2: "How effective is the dynamic programming based query planner in choosing physical plans?".

We have designed an experiment where the whole space of physical plans is explored exhaustively. For each physical plan, we accumulate the number of intermediate results when evaluating the physical plan. We compare the number of intermediate results of the chosen physical plan with that of all other physical plans. This approach was taken, as opposed to analyzing query evaluation time for example, because the number of intermediate results is independent of non-determinism in the runtime environment during query evaluation.

We use the procedure from Figure 6.2 to compare the chosen physical plan with the whole space of physical plans.

The result of this experiment for the LUBM dataset is shown in Table 6.6, and the result of this experiment for the Advogato dataset is shown in Table 6.6.

The results of this experiment on the LUBM dataset and the Advogato dataset are combined and grouped by the length of the paths to which the queries evaluate. These combined and grouped results are shown in Table 6.8.

$$ir(plan) = \text{sum of the number of results for each physical operator in } plan$$

$$best = min(ir(plan_i), \ldots, ir(plan_n))$$

$$worst = max(ir(plan_i), \ldots, ir(plan_n))$$

$$error(plan) = \frac{ir(plan) - best}{worst - best}$$

Figure 6.2: Schematic illustration of how we compare the number of intermediate results of the chosen physical plan, $plan$, with the whole space of physical plans, $[plan_1, \ldots, plan_n]$.

LUBM	Min plan	Max plan	Telepath plan	Error
Q1	3,059,339	3,187,176	3,059,339	0
Q2	1,900,541	2,195,712	1,900,541	0
Q3	6,032	115,142	6,032	0
Q4	3,394	17,080	3,394	0
Q5	750,958	2,775,277	787,127	0.02
Q6	1,210,066	4,391,635	1,210,066	0
Q7	207,562	984,212	207,562	0
Q8	738,677	5,623,456	738,677	0
Q9	2,120,823	4,406,681	2,120,823	0
Mean error				**0**

Table 6.6: Query planning experiment on the LUBM dataset showing the minimum and maximum number of intermediate results for the space of physical plans and how it compares to the chosen physical plan.

6.3.3 Exp3: Query execution evaluation

As described in Section 6.1, the goal of this experiment is to answer question G3: "How does the running time of query evaluation in *Telepath*, leveraging a disk-based *k-path* index, compare to the current state of the art graph database engines?".

We compare query execution time to retrieve the last result using our underlying disk-based *k-path* index with the open-source graph database engine Neo4j. Neo4j offers full transactional support, availability and scalability through distribution, and a declarative query language. We collect the query evaluation result via the Java API, that comes with the Neo4j distribution, by sending queries in its query language Cypher[1]. See Appendix B for the full list of Cypher queries for both the LUBM and the Advogato dataset. The experiment is conducted using the latest version of Neo4j available at the time, version 3.2.6.

Regarding the statistical analysis of this experiment, we obtain the timings of 20 runs for each query. We excluded 10% of the data from each end of the range for the set of results for each query, eliminating outliers due to non-determinism

[1]Cypher query language: https://neo4j.com/developer/cypher-query-language/.

Advogato	Min plan	Max plan	Telepath plan	Error
Q1	533,667	554,771	533,667	0
Q2	7,194,334	7,239,498	7,194,334	0
Q3	2,856,420	2,892,442	2,856,420	0
Q4	1,129,051	1,272,471	1,129,051	0
Q5	3,491,934	3,945,468	3,491,934	0
Q6	6,188,839	7,148,011	6,188,839	0
Q7	6,909,075	7,457,625	7,112,333	0.37
Q8	27,172,591	30,616,206	27,172,591	0
Q9	60,478,560	74,729,410	60,478,560	0
Mean error				**0.04**

Table 6.7: Query planning experiment on the Advogato dataset showing the minimum and maximum number of intermediate results for the space of physical plans and how it compares to the chosen physical plan.

Path length	Error
3	0
4	0.07
5	0
Mean error	**0.02**

Table 6.8: Combined error of the query planning experiment for both datasets, grouped by the length of the paths to which the queries evaluate.

in the runtime environment. We report here the mean of the remaining values.

Queries are evaluated in *Telepath* while benefitting a disk-based *k-path* index where $k = 1$, and where $k = 2$. Since all queries evaluate to paths of length 3 or more, query optimization including query planning and cardinality estimation will influence the query execution evaluation.

The results of this experiment on the LUBM dataset and the Advogato dataset are combined and grouped by the length of the paths to which the queries evaluate. These combined and grouped results are shown in Table 6.11.

The results of this experiment grouped by the size of the query result set can be found in Table 6.12, Figure 6.3, and Figure 6.4.

LUBM	Neo4j	$k = 1$		$k = 2$	
		Telepath	Speedup	Telepath	Speedup
Q1	2,122	1,645	1x	1,603	1x
Q2	2,262	1,370	2x	1,119	2x
Q3	370	21	18x	8	46x
Q4	346	13	27x	10	35x
Q5	4,603	863	5x	549	8x
Q6	7,186	819	9x	733	10x
Q7	1,061	502	2x	150	7x
Q8	4,563	592	8x	406	11x
Q9	5,019	1,763	3x	1,440	3x
Mean speedup			**8x**		**14x**

Table 6.9: Query evaluation experiment on the LUBM dataset showing the timings (ms) to retrieve the last query answer in Neo4j and *Telepath* with *k-path* indices available for $k = 1$, and for $k = 2$.

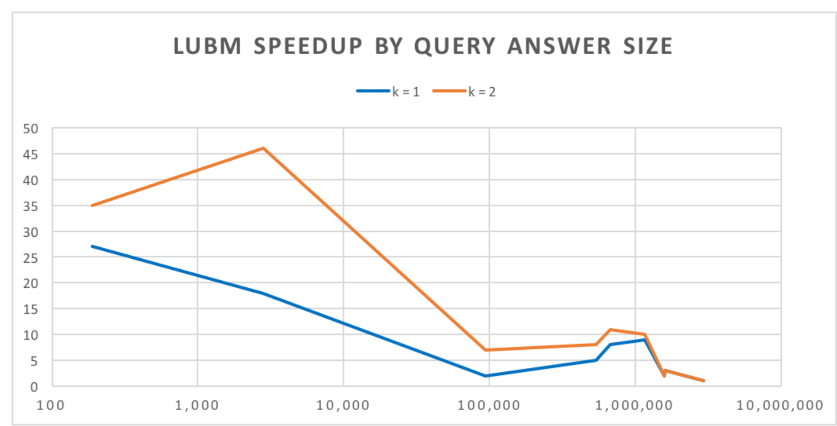


Figure 6.3: Speedup of query evaluation in *Telepath* compared to Neo4j for the LUBM dataset plotted against the size of the query result set.

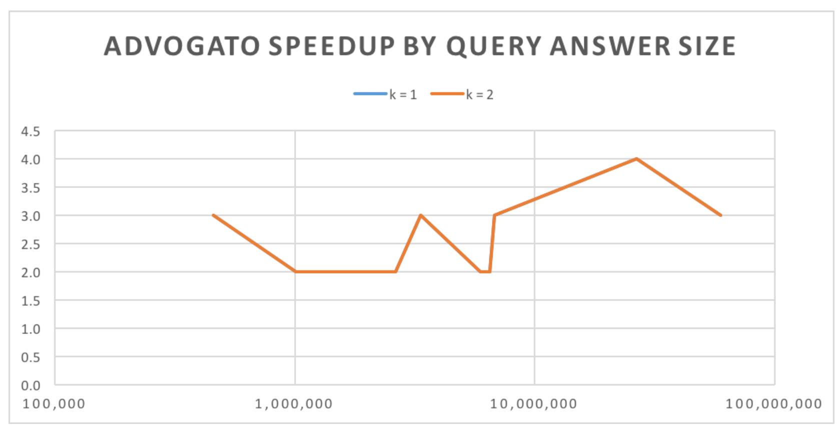


Figure 6.4: Speedup of query evaluation in *Telepath* compared to Neo4j for the Advogato dataset plotted against the size of the query result set.

Advogato	Neo4j	$k = 1$		$k = 2$	
		Telepath	Speedup	Telepath	Speedup
Q1	877	331	3x	311	3x
Q2	11,049	3,683	3x	3,904	3x
Q3	3,217	1,564	2x	1,498	2x
Q4	1,235	615	2x	634	2x
Q5	4,920	1,884	3x	1,820	3x
Q6	6,929	3,326	2x	3,282	2x
Q7	7,205	3,932	2x	3,793	2x
Q8	60,525	14,747	4x	14,801	4x
Q9	92,895	32,707	3x	32,715	3x
Mean speedup			**3x**		**3x**

Table 6.10: Query evaluation experiment on the Advogato dataset showing the timings (ms) to retrieve the last query answer in Neo4j and *Telepath* with *k-path* indices available for $k = 1$, and for $k = 2$.

Path length	LUBM speedup		Advogato speedup		Combined speedup	
	$k = 1$	$k = 2$	$k = 1$	$k = 2$	$k = 1$	$k = 2$
3	12x	21x	3x	3x	7x	12x
4	5x	8x	2x	2x	4x	6x
5	6x	7x	4x	4x	5x	5x
Mean speedup	8x	14x	3x	3x	**6x**	**9x**

Table 6.11: Combined speedup of query evaluation in *Telepath* compared to Neo4j for both datasets, grouped by the length of the paths to which the queries evaluate.

Result set	LUBM speedup		Advogato speedup		Combined speedup	
	$k = 1$	$k = 2$	$k = 1$	$k = 2$	$k = 1$	$k = 2$
0 - 100k	16x	29x	-	-	16x	29x
100k - 1m	7x	10x	3x	3x	5x	7x
1m - 10m	4x	4x	2x	2x	3x	3x
10m - 100m	-	-	4x	4x	4x	4x
Mean speedup	8x	14x	3x	3x	**6x**	**9x**

Table 6.12: Combined speedup of query evaluation in *Telepath* compared to Neo4j for both datasets, grouped by the size of the query result set.

6.4 Discussion

We see from the results of our experiments how the quality of cardinality estimates propagate to query execution times. Our relation statistics based cardinality estimator performs especially well on the LUBM dataset with only a mean error of **0.36**. This enables the dynamic programming based query planner to always choose a near optimal physical plan, as seen by a mean error of **0** for the LUBM dataset. Ultimately, we achieve a **14x** speedup for query execution times compared to `Neo4j`, a current state of the art graph database engine.

In contrast, cardinality estimation quality for the Advogato dataset results in a relatively worse mean error of **0.76**. While worse than with the LUBM dataset, such cardinality estimates still enable the query planner to achieve a mean error of **0.04** in choosing the optimal physical plan. However, compared to `Neo4j`, we achieve a **3x** speedup regarding query execution time.

If we group the query evaluation times by the size of the result set, compared to `Neo4j` we see that *Telepath* performs particularly well for smaller result sets, and performs decreasingly well if the result set grows. We achieve a mean speedup of **29x** for queries which evaluate to the smaller result sets of size $0 - 100k$. We see that the evaluation into larger result sets, and thus more intermediate results, is not able to leverage the disk-based *k-path* index in the same order of performance gain.

Our *k-path* index is based on the work of `PathDB` from [23]. Their study showed that their *k-path* index achieved at least a 2x speedup in query execution time, and in the best cases, a 8000x speedup. In contrast, *Telepath* provides at least a 2x speedup and at most a 46x speedup. This difference could be explained by a number of facts, e.g., *Telepath* ran their experiments on `Neo4j` version `3.2.6` while `PathDB` ran their experiments on version `2.3.0-M01`, *Telepath* does not have an implementation for the merge-join physical operator while `PathDB` does, *Telepath* did their experiments while having a *k-path* index available for k values up to 2, while `PathDB` did their experiments for k values up to 3, and *Telepath* evaluated queries during their experiments which have a length between $[3 - 5]$ while `PathDB` evaluated queries during their experiments which have a length between $[1 - 3]$.

However, with a combined mean error of **0.02** for the query planning experiment, we see that our dynamic programming based query planner always chooses a near optimal physical plan. We have shown that our dynamic programming based query planner algorithm chooses a physical plan in time polynomial in the size of the given regular path query (Equation 5.1), while the space of physical plans grows exponentially by the size of the given regular path query (Equation 5.3).

Chapter 7

Conclusions

Massive graph-structured data collections are omnipresent in modern data management scenarios such as social networks, linked open data, and chemical compound databases. Regular path queries for graph databases are eminently useful. However, these queries are difficult to optimize to evaluate efficiently.

The goal of this project has been to design, engineer and test a modular path-index based graph database engine with support for evaluation of regular path queries.

The main contributions can be summed up into the following four main areas: relation statistics based cardinality estimation where we provide a novel implementation to an existing concept from [29], dynamic programming based query planning where we provide a novel implementation to an existing concept from [18], end-to-end query evaluation using a disk-based *path* index where we use an existing implementation of a *path* index from [23]. However, we provide our own concept and implementation for end-to-end query evaluation while leveraging a *path* index. We designed, engineered, and tested the proposed approach.

Our work presents an end-to-end solution to evaluate a regular path query efficiently. The core of our end-to-end solution is the dynamic programming based query planner. The query planner chooses a physical plan in a bottom-up fashion by using cost estimates. Chosen physical plans for smaller subproblems are used to construct the physical plan for the global problem. Cost estimates are provided by using cardinality estimates for intermediate results. The cardinality estimates are produced through relation statistics maintained over the graph data. Switching to the end of the life-of-a-query, the *path* index, as engineered by [23], is used as an external library to retrieve (intermediate) query results. Intermediate results are mainly joined by our implementation of the hash join algorithm. This paragraph stated the main aspects of the mechanics of how *Telepath* efficiently produces query results for regular path queries.

Using one synthetic and one real-world dataset, we provide an empirical evaluation of the cardinality estimation, query planning, and query evaluation phases with a *path* index available for paths of length two. We see from the results of our experiments how the quality of cardinality estimates propagate towards the quality of query evaluation times. Our relation statistics based cardinality estimator generates estimates with a mean error of **0.56** on a scale from zero to one. This enables the dynamic programming based query planner to always choose a near optimal physical plan, as seen by a mean error of **0.02**. Ultimately, we achieve a **9x** mean speedup for query evaluation times compared to `Neo4j`, a current state of the art graph database engine.

If we group the query execution times by the size of the result set, compared to `Neo4j` we see that *Telepath* performs particularly well for smaller result sets, and

performs decreasingly well if the result set grows. We achieve a mean speedup of **29x** for queries which evaluate to smaller result sets of $0 - 100k$ tuples. We see that the evaluation into larger result sets, and thus more intermediate results, is not able to leverage the disk-based *path* index in the same order of performance gain.

However, with a combined mean error of **0.02** for the query planning experiment, we show that our dynamic programming based query planner always chooses a near optimal physical plan. We have shown that our dynamic programming based query planner algorithm chooses a physical plan in time polynomial in the size of the given regular path query (Equation 5.1), while the space of physical plans grows exponentially by the size of the given regular path query (Equation 5.3).

Regarding the engineering of *Telepath*, the implementation has been done in a modular and generic way to suit extensions and modifications by other researchers. We followed code quality standards which enforce readable, durable and maintainable source code. Furthermore, the implemented features have been unit tested which ensures a correctly working prototype when applying extensions and modifications.

Our *Telepath* prototype illustrates the performance gains that can be obtained in comparison to a current state of the art graph database engine, and it acts as a foundation for further research in the area of path-index based graph database engines due to its modular and generic design.

The source code has been made publicly available to contribute to the open-source community. The repository can be found on GitHub at https://github.com/giedomak/Telepath.

7.1 Future work

There are several natural directions for future study.

First, this work presented experiments which were performed on two diverse datasets, which establishes the performance benefits of our path-index based graph database engine. Future work can verify and establish the performance benefits by including more datasets.

Second, since the goal of this project has been to design and engineer an end-to-end solution, multiple components would benefit greatly when given dedicated research and engineering resources. Some “quick wins” include extending the physical library with the merge-join algorithm for concatenations, histogram-based cardinality estimation, heuristics based logical plan optimizations, and disk-based memory management improvements.

Third, unary operators currently don’t have an end-to-end working solution within *Telepath* as stated in Section 4.4. Although, the regular path query grammar is compatible with the Kleene star and the Kleene plus operator, as in the design of the dynamic programming based query planner.

Finally, one last idea worth mentioning is on the topic of handling heuristics for the union operator in logical plans. There might exist a scenario where a logical plan with its union operator pulled-up produces an effective physical plan. Also, vice versa for logical plans with its union operator pushed-down. One idea is to send both logical plans through the dynamic based query planner to produce a cost estimation for both of these logical plans.

Bibliography

[1] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George HL Fletcher, Aurélien Lemay, and Nicky Advokaat. gMark: schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering*, 29(4):856–869, 2017. 1

[2] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. Rewriting of regular expressions and regular path queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 194–204. ACM, 1999. 6

[3] Stavros Christodoulakis. On the estimation and use of selectivities in database performance evaluation. Technical report, Department of Computer Science, University of Waterloo, 1989. 10

[4] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. *SIGMOD Rec.*, 16(3):323–330, December 1987. 6

[5] George H.L. Fletcher, Jeroen Peters, and Alexandra Poulovassilis. Efficient regular path query evaluation using path indexes. In *EDBT*, pages 636–639. OpenProceedings.org, 2016. 11

[6] Peter Gassner. Query Optimization in the IBM DB2 Family. *Transformation*, 16(4):1–14, 1994. 25

[7] Gang Gou and Rada Chirkova. Efficiently querying large XML data repositories: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1381–1403, 2007. 11

[8] Andrey Gubichev and Manuel Then. Graph Pattern Matching - Do We Have to Reinvent the Wheel? In *GRADES*, pages 8:1–8:7. ACM, 2014. 11, 40

[9] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics*, 3(2-3):158–182, 2005. 40

[10] Peter Hilton and Jean Pedersen. Catalan numbers, their generalization, and their uses. *The Mathematical Intelligencer*, 13(2):64–75, 1991. 36

[11] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '12*, ASPLOS XVII, page 349, New York, NY, USA, 2012. ACM. 11

[12] Yannis E. Ioannidis. Query Optimization. *ACM Comput. Surv.*, 28(1):1–38, mar 1996. xi, 9, 10

[13] Yannis E. Ioannidis and Younkyung Kang. Randomized Algorithms for Optimizing Large Join Queries. *ACM SIGMOD Conf Management of Data, Atlantic City, NJ*, 19(2):312–321, 1990. 10

[14] KONECT. Trust network of the advogato online community dataset. http://konect.uni-koblenz.de/networks/advogato, apr 2017. 40, 41

[15] Robert Philip Kooi. *The optimization of queries in relational databases*. PhD thesis, Case Western Reserve University, Cleveland, OH, USA, 1980. 10

[16] Donald Kossmann and Konrad Stocker. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. *TODS*, 25(1):43–82, 2000. 25

[17] Michael V. Mannino, Paicheng Chu, and Thomas Sager. Statistical profile estimation in database systems. *ACM Comput. Surv.*, 20(3):191–221, 1988. 10

[18] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Vldb*, pages 930–941. VLDB Endowment, 2006. xi, 2, 18, 25, 27, 51

[19] Patrick O'Neil and Goetz Graefe. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record*, 24(3):8–11, 1995. 11

[20] Jeroen Peters. Regular path query evaluation using path indexes. Master's thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, 2015. 10

[21] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. Tamer Özsu. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: A User Survey. *ArXiv e-prints*, September 2017. 1

[22] P. Griffiths Selinger, M.M. Astrahan, D.D. Chamberlin, It. A. Lorie, and T.G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979. 10, 25

[23] Jonathan M. Sumrall. Path Indexing for Efficient Path Query Processing in Graph Databases. Master's thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, 2015. 1, 2, 18, 40, 49, 51

[24] Jonathan M. Sumrall, George H.L. Fletcher, Alexandra Poulovassilis, Johan Svensson, Magnus Vejlstrup, Chris Vest, and Jim Webber. Investigations on path indexing for graph databases. In *European Conference on Parallel Processing*, pages 532–544. Springer, 2016. 11

[25] Li Wang. On histograms for path selectivity estimation in graph data. Master's thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, 2017. xi, 43

[26] Adam Welc, Raghavan Raman, Zhe Wu, Sungpack Hong, Hassan Chafi, and Jay Banerjee. Graph analysis: do we have to reinvent the wheel? In *First International Workshop on Graph Data Management Experiences and Systems*, pages 7:1—-7:6. ACM, 2013. 11

[27] Bart G.J. Wolff, George H.L. Fletcher, and James J. Lu. An extensible framework for query optimization on TripleT-based RDF stores. In *CEUR Workshop Proceedings*, volume 1330, pages 190–196. OpenProceedings.org, 2015. 9

[28] Kam Fai Wong, Jeffrey Xu Yu, and Nan Tang. Answering XML queries using path-based indexes: A survey. *World Wide Web*, 9(3):277–299, 2006. 11

[29] Nikolay Yakovets. *Optimization of Regular Path Queries in Graph Databases*. PhD thesis, Department of Computer Science, York University, Canada, 2016. xi, xi, xi, 2, 32, 33, 51

Appendix A

Regular path query grammar

The following Listing defines the grammar for regular path queries as used by *Telepath*. The grammar is intended to be used by ANTLR[1] to parse user input.

```
/*
 * Regular path query (RPQ) grammar for ANTLR4.
 *
 * @author Giedo Mak
 * @author Nikolay Yakovets
 */

grammar RPQ;

/**
 * Parser rules
 * Each parser rule gets one of the following names: unaryExpression,
     binaryExpression, leaf or parenthesis
 */

query
    : query unaryOperator             # unaryExpression
    | query binaryOperator query      # binaryExpression
    | LABEL                           # leaf
    | '(' query ')'                   # parenthesis
    ;

unaryOperator
    : ( KLEENE_STAR | PLUS ) ;

binaryOperator
    : ( CONCATENATION | UNION ) ;

// Lexer rules

LABEL
    : ('!')? (CHARS+) ;

KLEENE_STAR
    : '*' ;

PLUS
    : '+' ;

CONCATENATION
    : '/' ;

UNION
    : '|' ;

CHARS
```

[1] ANTLR: http://www.antlr.org/.

```
    : 'A'..'Z'
    | 'a'..'z'
    | '0'
    | [1-9]
    | '\u00C0'..'\u00D6'
    | '\u00D8'..'\u00F6'
    | '\u00F8'..'\u02FF'
    | '\u0370'..'\u037D'
    | '\u037F'..'\u1FFF'
    | '\u200C'..'\u200D'
    | '\u2070'..'\u218F'
    | '\u2C00'..'\u2FEF'
    | '\u3001'..'\uD7FF'
    | '\uF900'..'\uFDCF'
    | '\uFDF0'..'\uFFFD'
    ;

WHITESPACE
    : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+ -> skip ;
```

Appendix B

Benchmark queries

This Appendix contains the formulated queries for both the LUBM and the Advogato dataset. Each set of regular path queries is also available as a set of Cypher queries.

B.1 LUBM dataset queries

B.1.1 Regular path queries

```
// ----- length = 3 -----

Q1: undergraduateDegreeFrom / !subOrganizationOf / !memberOf

Q2: advisor / teacherOf / !takesCourse

Q3: !headOf / worksFor / !subOrganizationOf

Q4: !headOf / worksFor / subOrganizationOf

// ----- length = 4 -----

Q5: advisor / teacherOf / !takesCourse / advisor

Q6: undergraduateDegreeFrom / !subOrganizationOf / !memberOf / advisor

Q7: !worksFor / teacherOf / !takesCourse / advisor

// ----- length = 5 -----

Q8: !teacherOf / undergraduateDegreeFrom / !subOrganizationOf / !
    memberOf / advisor

Q9: advisor / teacherOf / !takesCourse / advisor / teacherOf
```

Listing B.1: Regular path queries intended for the LUBM dataset.

B.1.2 Cypher queries

```
// ----- length = 3 -----

Q1: MATCH (a)-[:undergraduateDegreeFrom]->(b)<-[:subOrganizationOf]-(c
    )<-[:memberOf]-(d) RETURN COUNT(a)
```

```
Q2: MATCH (a)-[:advisor]->(b)-[:teacherOf]->(c)<-[:takesCourse]-(d)
    RETURN COUNT(a)

Q3: MATCH (a)<-[:headOf]-(b)-[:worksFor]->(c)<-[:subOrganizationOf]-(d
    ) RETURN COUNT(a)

Q4: MATCH (a)<-[:headOf]-(b)-[:worksFor]->(c)-[:subOrganizationOf]->(d
    ) RETURN COUNT(a)

// ----- length = 4 -----

Q5: MATCH (a)-[:advisor]->(b)-[:teacherOf]->(c)<-[:takesCourse]-(d)-[:
    advisor]->(e) RETURN COUNT(a)

Q6: MATCH (a)-[:undergraduateDegreeFrom]->(b)<-[:subOrganizationOf]-(c
    )<-[:memberOf]-(d)-[:advisor]->(e) RETURN COUNT(a)

Q7: MATCH (a)<-[:worksFor]-(b)-[:teacherOf]->(c)<-[:takesCourse]-(d)
    -[:advisor]->(e) RETURN COUNT(a)

// ----- length = 5 -----

Q8: MATCH (a)<-[:teacherOf]-(b)-[:undergraduateDegreeFrom]->(c)<-[:
    subOrganizationOf]-(d)<-[:memberOf]-(e)-[:advisor]->(f) RETURN
    COUNT(a)

Q9: MATCH (a)-[:advisor]->(b)-[:teacherOf]->(c)<-[:takesCourse]-(d)-[:
    advisor]->(e)-[:teacherOf]->(f) RETURN COUNT(a)
```

Listing B.2: Cypher queries intended for the LUBM dataset.

B.2 Advogato dataset queries

B.2.1 Regular path queries

```
// ----- length = 3 -----

Q1: apprentice / apprentice / apprentice

Q2: journeyer / journeyer / journeyer

Q3: master / master / master

Q4: apprentice / journeyer / master

// ----- length = 4 -----

Q5: apprentice / apprentice / apprentice / !journeyer

Q6: apprentice / journeyer / !apprentice / master

Q7: master / apprentice / !master / journeyer

// ----- length = 5 -----

Q8: apprentice / apprentice / apprentice / apprentice / apprentice
```

```
Q9: apprentice / journeyer / !master / !apprentice / master
```

Listing B.3: Regular path queries intended for the Advogato dataset.

B.2.2 Cypher queries

```
// ----- length = 3 -----

Q1: MATCH (a)-[:apprentice]->(b)-[:apprentice]->(c)-[:apprentice]->(d)
     RETURN COUNT(a)

Q2: MATCH (a)-[:journeyer]->(b)-[:journeyer]->(c)-[:journeyer]->(d)
    RETURN COUNT(a)

Q3: MATCH (a)-[:master]->(b)-[:master]->(c)-[:master]->(d) RETURN
    COUNT(a)

Q4: MATCH (a)-[:apprentice]->(b)-[:journeyer]->(c)-[:master]->(d)
    RETURN COUNT(a)

// ----- length = 4 -----

Q5: MATCH (a)-[:apprentice]->(b)-[:apprentice]->(c)-[:apprentice]->(d)
    <-[:journeyer]-(e) RETURN COUNT(a)

Q6: MATCH (a)-[:apprentice]->(b)-[:journeyer]->(c)<-[:apprentice]-(d)
    -[:master]->(e) RETURN COUNT(a)

Q7: MATCH (a)-[:master]->(b)-[:apprentice]->(c)<-[:master]-(d)-[:
    journeyer]->(e) RETURN COUNT(a)

// ----- length = 5 -----

Q8: MATCH (a)-[:apprentice]->(b)-[:apprentice]->(c)-[:apprentice]->(d)
    -[:apprentice]->(e)-[:apprentice]->(f) RETURN COUNT(a)

Q9: MATCH (a)-[:apprentice]->(b)-[:journeyer]->(c)<-[:master]-(d)<-[:
    apprentice]-(e)-[:master]->(f) RETURN COUNT(a)
```

Listing B.4: Cypher queries intended for the Advogato dataset.

Appendix C

Engineering Telepath

C.1 Directory structure

```
.
`-- src
    |-- main
    |   |-- java
    |   |   `-- com
    |   |       `-- github
    |   |           `-- giedomak
    |   |               `-- telepath
    |   |                   |-- cardinalityestimation
    |   |                   |   `-- synopsis
    |   |                   |-- costmodel
    |   |                   |-- datamodels
    |   |                   |   |-- graph
    |   |                   |   |-- integrations
    |   |                   |   |-- plans
    |   |                   |   |   `-- utilities
    |   |                   |   `-- stores
    |   |                   |-- evaluationengine
    |   |                   |-- kpathindex
    |   |                   |   `-- utilities
    |   |                   |-- memorymanager
    |   |                   |   `-- spliterator
    |   |                   |-- physicaloperators
    |   |                   |-- planner
    |   |                   |   `-- enumerator
    |   |                   |-- staticparser
    |   |                   |   `-- rpq
    |   |                   `-- utilities
    |   `-- resources
    |       `-- antlr4
    |           |-- rpq
    |           |   `-- examples
    |           `-- sparql
    |               `-- examples
    `-- test
        |-- java
        |   `-- com
        |       `-- github
        |           `-- giedomak
        |               `-- telepath
        |                   |-- cardinalityestimation
        |                   |   `-- synopsis
        |                   |-- datamodels
        |                   |   `-- plans
        |                   |       `-- utilities
        |                   |-- evaluationengine
        |                   |-- integrationtests
```

```
           |                    |-- kpathindex
           |                    |   `-- utilities
           |                    |-- memorymanager
           |                    |-- physicaloperators
           |                    |-- planner
           |                    |   `-- enumerator
           |                    |-- staticparser
           |                    `-- utilities
           `-- resources
               `-- mockito-extensions
```

Listing C.1: Directory structure of *Telepath*

C.2 Interface example

As an example, the following Listing shows the interface for physical operators. This interface defines that any implementation of a physical operator must implement the `physicalPlan` property, and provide implementations for the `evaluate()` and `cost()` functions.

```
/**
 * Interface for physical operators.
 *
 * @property physicalPlan The physical plan holds information
     regarding the relations on which to operate.
 */
interface PhysicalOperator {

    val physicalPlan: PhysicalPlan

    /**
     * Evaluates the physical operator and produces a PathStream.
     *
     * @return PathStream with the results of the evaluation.
     */
    fun evaluate(): PathStream

    /**
     * Calculates the cost of the physical operation.
     *
     * @return The cost of the physical operation.
     */
    fun cost(): Long
}
```

Listing C.2: Interface for physical operators

C.3 Documentation

The source code is well documented using Javadoc[1] style comments in the source code. The generated HTML of the source code documentation is publicly available at https://giedomak.github.io/Telepath/telepath/, hosted by GitHub pages[2].

The repository also contains four elaborative documents, e.g., a readme, a contributing guide, the internal behavior of the query planner guide, and a guide on how to add an extra physical operator. These elaborative documents can be found in the repository hosted by GitHub, and in the following four subsections.

C.3.1 Repository readme

See Figure C.1 for the readme of the *Telepath* repository. This guide can also be found at https://github.com/giedomak/Telepath.

C.3.2 Repository contributing guide

See Figure C.4 for the contributing guide of the *Telepath* repository. This guide can also be found at https://github.com/giedomak/Telepath/blob/master/CONTRIBUTING.md.

C.3.3 Query planner guide

See Figure C.6 for the guide on the internal behavior of the query planner of *Telepath*. This guide can also be found at https://github.com/giedomak/Telepath/tree/master/src/main/java/com/github/giedomak/telepath/planner.

C.3.4 Guide on adding an extra physical operator

See Figure C.11 for the guide on how to add an extra physical operator to *Telepath*. This guide can also be found at https://github.com/giedomak/Telepath/tree/master/src/main/java/com/github/giedomak/telepath/physicaloperators.

[1] Javadoc is a tool for generating API documentation in HTML format from comments in source code.

[2] GitHub pages: https://pages.github.com/.

TelepathDB

build passing
codacy A
code climate 4.0
codebeat A
codecov 78%

Massive graph-structured data collections are ubiquitous in contemporary data management scenarios such as social networks, linked open data, and chemical compound databases.

The selection and manipulation of paths forms the core of querying graph datasets. Path indexing techniques can speed up this core functionality of querying graph datasets.

We propose a path-index based graph database engine.

Documentation

The documentation can be found here and a schematic overview of the architecture can be found here.

Life of a Query

This section describes the essence of the life of a query within TelepathDB. Each heading contains links to its docs, test and source. In most cases, the test will give a clear insight into what each specific module produces.

1. **Query input**

   The user gives a regular path query as input. For example:

   ```
   a/(b/c)
   ```

   Where `a`, `b` and `c` are edge labels, and `/` is interpreted as the concatenation logical operator.

Figure C.1: The readme of the *Telepath* repository page 1.

2. **Parse the input** (docs) (test) (source)

   The query input is parsed into our internal representation of a logical plan. Our internal representation uses a tree datastructure:

```
CONCATENATION
  /       \
 a    CONCATENATION
        /   \
       b     c
```

3. **Generate the cheapest physical plan** (docs) (test) (source)

   Our planner uses the `DPsize` algorithm as inspiration, which calculates the cheapest physical plan in a bottom-up fashion.

   Since this phase is one of the main contributions, an in-depth explanation can be found here.

```
INDEX_LOOKUP
  /  |  \
 a   b   c
```

4. **Evaluate the physical plan**

   The physical plan is evaluated in a bottom-up fashion. All intermediate results are materialized through our MemoryManager (docs) (test) (source).

   Using PathDB to gather the paths satisfying our query:

```
kPathIndex.search(
        PathPrefix(
                physicalPlan.pathIdOfChildren()
        )
)
```

5. **Visualize results**

   At the time of writing, results will be shown to the user through a command-line interface.

Figure C.2: The readme of the *Telepath* repository page 2.

```
TelepathDB: >>>>> Results:
TelepathDB: Path(pathId=9, nodes=[Node(id=10), Node(id=12), Node(id=14)])
TelepathDB: Path(pathId=9, nodes=[Node(id=10), Node(id=12), Node(id=8772)])
TelepathDB: Number of results: 2, after 5 ms
TelepathDB: ---------------------------
```

Want to contribute?

The contributing guide
is a good place to start. If you have questions, feel free to ask.

Authors

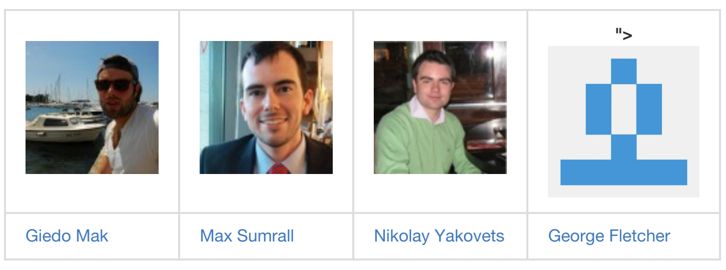


Figure C.3: The readme of the *Telepath* repository page 3.

Contributing

First off, thank you for considering contributing to TelepathDB. It's people like you that make TelepathDB such a great system.

1. Where do I go from here?

If you've noticed a bug or have a question, search the issue tracker to see if someone else in the community has already created a ticket. If not, go ahead and make one!

2. Fork & create a branch

If this is something you think you can fix, then fork TelepathDB and create a branch with a descriptive name.

A good branch name would be (where issue #325 is the ticket you're working on):

```
git checkout -b 325-add-japanese-translations
```

3. Get the test suite running

Make sure you're using Java `1.8` and you have installed at least version `3.5` of Maven.

Now you should be able to run the entire test suite using:

```
mvn clean test
```

4. Implement your fix or feature

At this point, you're ready to make your changes! Feel free to ask for help; everyone is a beginner at first 😸.

Available getting started guides:

Figure C.4: The contributing guide of the *Telepath* repository page 1.

- Implementing a new physical operator

5. Make a Pull Request

At this point, you should switch back to your master branch and make sure it's
up to date with Active Admin's master branch:

```
git remote add upstream git@github.com:giedomak/TelepathDB.git
git checkout master
git pull upstream master
```

Then update your feature branch from your local copy of master, and push it!

```
git checkout 325-add-japanese-translations
git rebase master
git push --set-upstream origin 325-add-japanese-translations
```

Finally, go to GitHub and make a Pull Request :D

Travis CI will run the test suite and other integrations like codeclimate will analyse the code quality.
We care about quality, so your PR won't be merged until the tests pass and the quality of the PR is good enough.

Figure C.5: The contributing guide of the *Telepath* repository page 2.

Planner

This document describes how the planner calculates the cheapest physical plan for a given logical plan.

1. Flatten into multi-children tree (docs) (test) (source)

Logical plans are flattened to prepare them for the subtree generator.

Given:

```
    CONCATENATION
       /    \
      a     CONCATENATION
               /    \
              b      c
```

Output:

```
   CONCATENATION
      /  |  \
     a   b   c
```

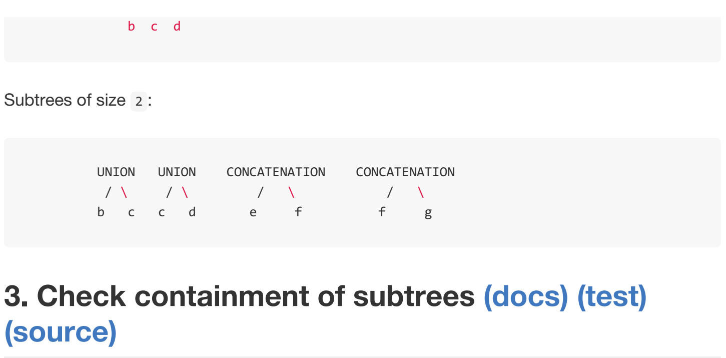
2. Generate subtrees of a given size (docs) (test) (source)

Let's say we are trying to calculate the cheapest physical plan for a plan with size `2`. Then we are generating all subtrees of size `1`, and check if we can combine them. These smaller subtrees have its cheapest physical plan already calculated, so we'll want to re-use those.

Given:

```
       CONCATENATION
      /  |     |  |  \
     a  UNION  e  f   g
         / | \
```

Figure C.6: The query planner guide page 1.



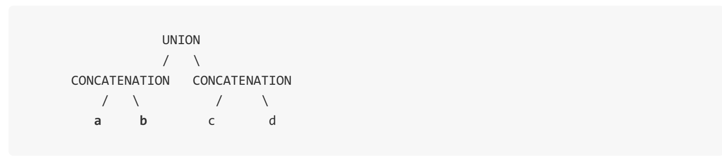
Subtrees of size `2`:

```
UNION   UNION    CONCATENATION    CONCATENATION
 / \     / \         /   \            /   \
b   c   c   d       e     f          f     g
```

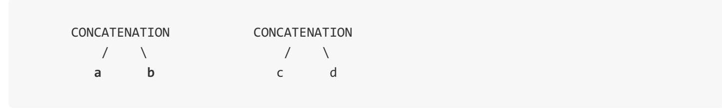
3. Check containment of subtrees (docs) (test) (source)

When two subtrees are contained in the logical plan through any operator, we calculate the cheapest physical plan for the combination of those two subtrees concatenated by the operator.

Given this logical plan:



Given `subtree1` and `subtree2`:



`subtree1` and `subtree2` are contained in the logical plan through the `UNION` operator.

Second example:

Given this logical plan:

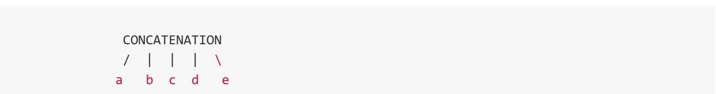
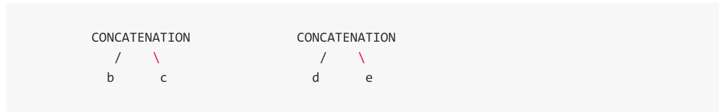


Figure C.7: The query planner guide page 2.

Given `subtree1` and `subtree2`:



`subtree1` and `subtree2` are contained in the logical plan through the `CONCATENATION` operator.

4. Enumerate operators (docs) (test) (source)

When two subtrees are contained through an operator in the logical plan, we'll calculate the cheapest physical plan for their combination. Remember we already know the cheapest physical plans for both subtrees.

As an example, let's say we've got two subtrees contained through the `CONCATENATION` operator. We enumerate the logical operator into hash-join, nested-loop-join and index-lookup.

Given these two trees and the CONCATENATION operator with a k-value greater than or equal to `4`:

```
INDEX_LOOKUP     INDEX_LOOKUP
  /     \          /     \
 a       b        c       d
```

Expected:

5. Cardinality estimation (docs) (test) (source)

When we are dealing with intermediate results, we need the estimated cardinality of these

Figure C.8: The query planner guide page 3.

intermediate results to calculate the cost.

If we are dealing with only concatenations, we can use our SynopsisCardinalityEstimation. This cardinality estimator will track graph statistics into a synopsis. It will hold information on all concatenations up to `k = 2`.

Using this synopsis we can estimate the cardinality of paths where `k = 3`, by using the synopsis for `k = 1` and `k = 2`.

```
// See if we got one of these after flattening:
//
//              HASH_JOIN
//              /       \
//     INDEX_LOOKUP   INDEX_LOOKUP
//       /  |  \        /   \
//      a   b   c      d     e      <--- EDGES
if (clone.operator in PhysicalOperator.JOIN_OPERATORS && clone.height() == 2) {

    val edges: List<Edge> = clone.children.flatMap { it.children.map { it.leaf!! } }

    // We can get | T r/l1 | from our Synopsis.
    var cardinality = synopsis.pairs(Pair(edges[0], edges[1])).toFloat()

    // | T r/l1/l2 | = | T r/l1 | * ( l1/l2.two / l1.in )
    for (index in 2 until edges.size) {

        val l1 = edges[index - 1]
        val l2 = edges[index]

        cardinality *= synopsis.two(Pair(l1, l2)) / synopsis.`in`(l1).toFloat()

    }

    // Return the result
    return cardinality.toLong()
}
```

6. Costing physical plans

Each physical operator has a cost associated to it which depends on the cardinality of the sets it operates on.

For example, the cost of hash-join is `2 * (M + N)`. Where `M` is the cardinality of set 1, and `N` is the cardinality of set 2.

```
/**
```

Figure C.9: The query planner guide page 4.

```
 * Cost of Hash-join.
 */
override fun cost(): Long {

    // The cost to produce results, i.e. 2 * (M + N)
    val myCost = 2 * (firstChild.cardinality + lastChild.cardinality)

    // Our input sets might be intermediate results, so take their cost into account.
    val cost1 = firstChild.cost()
    val cost2 = lastChild.cost()

    // Overflow check
    if (myCost == Long.MAX_VALUE || cost1 == Long.MAX_VALUE || cost2 == Long.MAX_VALUE) ret

    return myCost + cost1 + cost2
}
```

7. Save the cheapest physical plan

Once each enumerated physical plan has been costed, we save the cheapest physical plan. Since we work in a bottom-up fashion, after all iterations, we will have calculated the cheapest physical plan for the given logical plan.

Figure C.10: The query planner guide page 5.

Adding a new physical operator

To add a new physical operator, we need a couple of things:

- Implement the PhysicalOperator interface.
- Enumeration from the logical operator into our new physical operator.
- Cardinality estimation of the result set from our new physical operator.
- Cost of the evaluation of our new physical operator.
- Evaluation of our new physical operator.

PhysicalOperator implementation (docs) (source)

The companion object from the PhysicalOperator class is responsible for the mapping from physical operator constants to the actual implementation. Implement those here.

The actual implementation is described in the Costing an Evaluation section.

Code snippet of the symbolic mapping:

```
companion object {

    // ------ CONSTANTS ------

    const val LEAF = 0

    const val INDEX_LOOKUP = 1

    const val HASH_JOIN = 2
    const val NESTED_LOOP_JOIN = 3

    const val UNION = 4

    // ------ COLLECTIONS -------

    val JOIN_OPERATORS = listOf(HASH_JOIN, NESTED_LOOP_JOIN)

    // ------ FUNCTIONS -------

    /**
     * Convert the operators constants to an actual [PhysicalOperator] instance.
     *
     * @param physicalPlan which holds the operator constant.
     * @return The PhysicalOperator instance which has knowledge of the physical plan.
     */
    fun getPhysicalOperator(physicalPlan: PhysicalPlan): PhysicalOperator? {

        return when (physicalPlan.operator) {
```

Figure C.11: The guide on adding an extra physical operator to *Telepath* page 1.

```
                LEAF -> null

                INDEX_LOOKUP -> IndexLookup(physicalPlan)

                HASH_JOIN -> HashJoin(physicalPlan)
                NESTED_LOOP_JOIN -> NestedLoopJoin(physicalPlan)

                UNION -> Union(physicalPlan)

                else -> TODO("Gotta catch em all")
            }
        }
    }
```

Enumerate operator (docs) (test) (source)

This example combines two physical plans by enumerating over the applicable physical operators for the `CONCATENATION` logical operator:

```
             INDEX_LOOKUP       INDEX_LOOKUP
               /     \            /     \
              a       b          c       d
```

Expected:

```
          INDEX_LOOKUP              HASH_JOIN               NESTED_LOOP_JOIN
           / |  |  \                 /    \                   /       \
           a  b  c   d       INDEX_LOOKUP INDEX_LOOKUP    INDEX_LOOKUP INDEX_LOOKUP
                               /     \      /    \          /     \      /    \
                              a       b    c      d        a       b    c      d
```

The code snippet making this possible:

```
private fun enumerateConcatenation(tree1: PhysicalPlan, tree2: PhysicalPlan): Sequence<Phys

    val physicalPlans = mutableListOf<PhysicalPlan>()

    // Check if an INDEX_LOOKUP is applicable.
    val plan = tree1.merge(tree2, PhysicalOperator.INDEX_LOOKUP).flatten()

    // If the height of this tree is 1 (max number of edges to any leaf), AND the number of
    // is smaller or equal to the k-value of our index, we can do an INDEX_LOOKUP!
    if (plan.height() == 1 && plan.children.size <= plan.query.telepathDB.kPathIndex.k) {
```

Figure C.12: The guide on adding an extra physical operator to *Telepath* page 2.

```
        physicalPlans.add(plan)
    }

    // Don't forget to enumerate all the JOIN_OPERATORS
    PhysicalOperator.JOIN_OPERATORS.forEach {
        physicalPlans.add(tree1.merge(tree2, it))
    }

    return physicalPlans.asSequence()
}
```

Cardinality estimation (docs) (test) (source)

As you can see in the current implementation, cardinality estimates for `JOIN_OPERATORS` will just take the max cardinality of its two datasets. The `UNION` physical operator will get you the sum of the cardinalities of its children.

Code snippet:

```
/**
 * Returns the cardinality of a given physicalPlan.
 *
 * This method will recursively calculate the cardinality for its children in order to get
 * for the root.
 *
 * @param physicalPlan The root of the tree for which we want to get the cardinality.
 * @return The cardinality of the given physicalPlan.
 */
override fun getCardinality(physicalPlan: PhysicalPlan): Long {

    return when (physicalPlan.operator) {

        PhysicalOperator.INDEX_LOOKUP -> getCardinality(physicalPlan.pathIdOfChildren())

        in PhysicalOperator.JOIN_OPERATORS -> {
            val d1 = getCardinality(physicalPlan.children.first())
            val d2 = getCardinality(physicalPlan.children.last())
            Math.max(d1, d2)
        }

        PhysicalOperator.UNION -> {
            getCardinality(physicalPlan.children.first()) + getCardinality(physicalPlan.chi
        }

        else -> TODO("You forgot one!")
    }
}
```

Figure C.13: The guide on adding an extra physical operator to *Telepath* page 3.

Costing & Evaluation

Costing and Evaluation are both delegated to the PhysicalOperator implementation to maintain a more object-oriented approach.

See the hash-join implementation as a reference. (docs) (test) (source)

Code snippet of the hash-join implementation:

```
/**
 * Hash-join physical operator.
 *
 * @property physicalPlan The physical plan holds information regarding the sets on which t
 * @property firstChild The first set of data to operate on, which is a [PhysicalOperator]
 * @property lastChild The last set of data to operate on, which is a [PhysicalOperator] it
 */
class HashJoin(override val physicalPlan: PhysicalPlan) : PhysicalOperator {

    /**
     * Evaluate the hash-join.
     *
     * @return A stream with the concatenated paths.
     */
    override fun evaluate(): PathStream {
        return OpenHashJoin(
                firstChild.evaluate(),
                lastChild.evaluate(),
                physicalPlan.query.telepathDB
        ).evaluate()
    }

    /**
     * Cost of Hash-join.
     */
    override fun cost(): Long {

        // The cost to produce results, i.e. 2 * (M + N)
        val myCost = 2 * (firstChild.cardinality() + lastChild.cardinality())

        // Our input sets might be intermediate results, so take their cost into account.
        val intermediateResultsCost = firstChild.cost() + lastChild.cost()

        return myCost + intermediateResultsCost
    }
}
```

Figure C.14: The guide on adding an extra physical operator to *Telepath* page 4.