A Statistical Model to Estimate Result Cardinality in a Graph Database

Bachelor's Thesis

submitted by

Moritz Renftle

at the

Universität Konstanz

Faculty of Sciences

Department of Computer Science

1st Reviewer: Prof. Dr. Michael Grossniklaus

2nd Reviewer: Prof. Dr. Marc H. Scholl

Konstanz, 2017

A Statistical Model to Estimate Result Cardinality in a Graph Database

Abstract

In this thesis we address two issues in the field of graph databases.
Firstly, the relational algebra is a general and precise mathematical language to design and analyze relational database systems. We argue that, the graph database community needs a commonly accepted graph query algebra for the same purposes. We define a graph query algebra that can be used to express *pattern matching* queries with constraints on relationship types, node labels and relationship uniqueness. Our algebra covers most of the pattern matching available in the popular graph query language Cypher.
Secondly, current graph databases often do not keep their promise of outperforming a mapping to a relational database, because their query optimizers are less sophisticated. As a first step towards better graph query optimizers, we develop a new model to estimate result cardinality in a graph database based on our new graph query algebra. Our model uses two helper data structures to inform itself about the distribution of node labels in the database, avoiding the estimation errors of models assuming independence between all node labels. In addition, the model performs well even if there are strong dependencies between labels and the presence of relationships in the database. In an experimental analysis we demonstrate the superiority of our model compared to Neo4j's current estimation model.

Contents

1. Introduction

1.1. Motivation

Graph query languages have become very popular in some domains recently, e.g., in the analysis of social networks. In those domains people are interested in queries that relate many different entities in a complex pattern. Such queries are often difficult to express and understand for users in terms of tables, rows and columns, as e.g, in SQL. However they can be directly translated into a graph with (labeled) nodes for the entities and (typed) edges for the relationships. As a result, there is growing interest in the employment of graph query languages.

Graph query languages can be used on top of a relational database, using a suitable mapping from graph queries to relational queries. This approach has the advantage of reusing all the performance optimizations developed for relational databases during the last decades.

Another approach is to use a graph database. Graph databases use graph structures not only for querying, but also for storing and processing of the data. Normally this means that relationships and entities are stored separately and that physical pointers are used to reference entities.

Several empirical studies were conducted recently to compare the performance of graph databases with mappings to relational databases. Gubichev and Then [5] demonstrate that a mappping to the relational database Virtuoso outperforms the graph databases Neo4j and DEX/Sparksee for a set of pattern matching queries. Hölsch et al. [7] show that two commercial relational databases using a mapping are faster than Neo4j for analytical queries and for pattern matching queries with relationship type constraints. These test results indicate that suitable mappings to relational databases currently outperform graph databases.

The worse performance of graph databases can partly be attributed to their inferior query optimizers. For instance, the optimizer of the Neo4j graph database fails to choose a good execution plan for some simple queries [6]. Consequently, there is a need for better query optimizers for graph databases.

In a currently ongoing research effort at the University of Konstanz, we develop a new optimizer for the Neo4j graph database. This optimizer will be based on the Cascades optimizer framework defined by Graefe [3].

The Cascades framework generates execution plans using equivalence rules on a query algebra. Therefore, we have to define a query algebra for graph databases. Hölsch et al. [6] have already defined two operators of this algebra. In this thesis, we formally define the missing operators and prove that our set of operators is complete and well-defined.

In addition, the Cascades framework requires robust estimations of the result cardinalities of each operator of this algebra. In this thesis, we propose such estimations and evaluate them in the Neo4j graph database.

1.2. Contributions

We make two contributions in the field of graph query optimization.

1.2.1. A Graph Query Algebra

Firstly, we define a graph query space in terms of subgraph homomorphisms. Together with a set of query operators, the query space forms a graph query algebra. This query algebra is a formalization of a subset of the very popular graph query language *Cypher* [1]. The graph algebra can be used in a graph DBMS for the same kind of tasks as the relational algebra in a relational DBMS, e.g. for search space exploration and statistical analysis.

1.2.2. A Statistical Model to Estimate Result Cardinality

Secondly, we propose a statistical model to estimate the result cardinalities of the operators of the graph algebra. We implement this model in the Neo4j graph database.

In contrast to Neo4j's own estimation model, our model does not globally assume independence of node labels. Using two helper data structures, it also yields sensible estimations if node labels are disjoint or in a sublabel relation. Additionally, the model is able to deal with strong dependencies between node labels and the presence of relationships between nodes.

Because of missing statistics in Neo4j and the early stage of development of our model, the query space covered by the model is currently limited to (Cypher) single pattern queries (CSP queries), that are cycle-free and only have one component. Once these problems are solved, the model can be extended to cover the whole query space.

A statistical analysis at the end of this thesis shows the benefits of our model compared with the current estimation model of Neo4j.

[1] `http://www.opencypher.org/`

2. The Query Algebra

In this chapter, we formally define a graph query space in terms of subgraph homomorphisms. Combined with a set of query operators, this query space forms a graph query algebra. We show that each operator is well-defined and give a complete set of operators.

2.1. Query Results

Definition 2.1.1 (Database graph)**.** In this thesis, we consider graph databases that are able to store the information of a directed, labeled and typed multigraph G (loops are allowed). We call G the database graph and define it as follows:
$G = (V, R, \lambda, \mathtt{l}, \mathtt{t})$, where

- V is a finite set of nodes
- R is a finite set of relationships
- $\mathcal{D}_L$ is a set of node labels
- $\mathcal{D}_R$ is a set of relationship types
- $\lambda : R \to V \times V$ is a function assigning nodes to relationships
- $\mathtt{l} : V \to \mathcal{P}(\mathcal{D}_L)$ is a function assigning labels to nodes ($\mathcal{P}(X)$ denotes the powerset of a finite set X)
- $\mathtt{t} : E \to \mathcal{D}_R$ is a function assigning relationship types to relationships.

Figure 2.1 shows an example database graph from a fictitious social network. The function $\mathtt{l}$ is encoded by writing all labels from $\mathtt{l}(v)$ next to the node v in the graph, prefixed by a colon. The function $\mathtt{t}$ is encoded by writing $\mathtt{t}(r)$ after each relationship r in the graph, separated by a colon. In the example graph, nodes and relationships are identified by natural numbers. It is required that nodes and relationships are disjoint. Therefore, we write x_n to refer to the node identified by the number x and x_r to refer to the relationship identified by x.

Remark 1. In this thesis, we are only interested in the statistical information derivable from node labels and relationship types. We ignore that most graph databases additionally provide a way of storing properties at nodes.

Node properties can be added to our graph model and to our query algebra without any interference in the future.

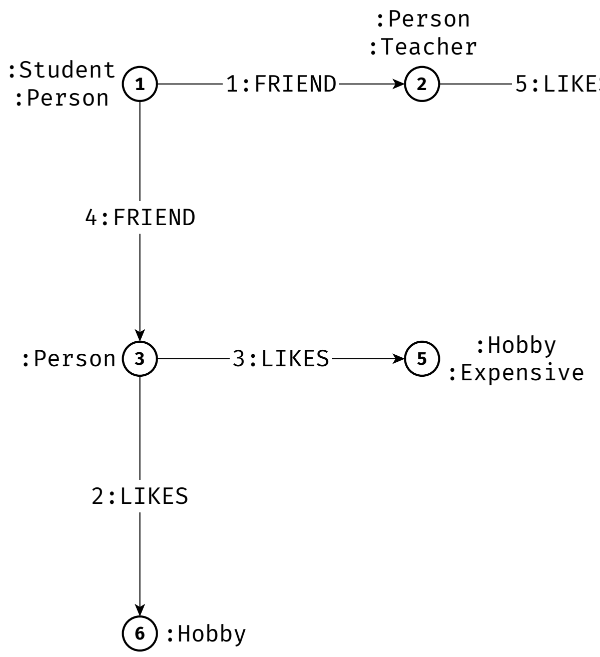


Figure 2.1.: A directed, labeled and typed multigraph.

In a graph database system, queries return sets of subgraphs of the database G. A subgraph can be defined in various ways. In its most general form, a subgraph is a sequence of nodes of the database, together with a sequence of relationships between these nodes. Instead of natural numbers, we use a domain of query variables $\mathcal{X}$ to enumerate the nodes and the relationships.

Definition 2.1.2 (Subgraph)**.** Consider a domain of query variables $\mathcal{X}$. Let X be a finite subset of this domain. A subgraph of the database G is a function $\mu : X \to V \cup R$ such that

$$\forall r \in R \;\; \forall v_1 \in X \;\; (\mu(v_1) = r \;\Rightarrow\; \exists v_2, v_3 \in X \;\; \lambda(r) = (\mu(v_2), \mu(v_3)))$$

For $\mu(v) = x$ we say that x is *matched* by the variable v in the subgraph μ.
We use the following set notation to write a subgraph $\mu : \{v_1, \ldots, v_n\} \to V \cup R$:

$$\mu = \{v_1 \mapsto \mu(v_1), v_2 \mapsto \mu(v_2) \ldots, v_n \mapsto \mu(v_n)\}$$

In a *query result* we want to collect all subgraphs having a particular *pattern*. This pattern is defined as another graph, whose nodes and relationships consist of query variables. The subgraphs having this pattern are all subgraphs that are a graph homomorphism from the pattern graph to the database graph. Our notion of graph homomorphism includes constraints on labels, relationship types and on relationship uniqueness.

Component	Space complexity
V_ρ	$O(\lvert V_\rho\rvert)$
R_ρ	$O(\lvert R_\rho\rvert)$
λ_ρ	$O(\lvert R_\rho\rvert)$
$\mathtt{l}_\rho$	$O(\lvert V_\rho\rvert \cdot \lvert\mathcal{D}_L\rvert)$
$\mathtt{t}_\rho$	$O(\lvert R_\rho\rvert)$
$\mathtt{u}_\rho$	$O(\binom{\lvert R_\rho\rvert}{2})$
Total:	$O(\binom{\lvert R_\rho\rvert}{2} + \lvert V_\rho\rvert \cdot \lvert\mathcal{D}_L\rvert)$

Table 2.1.: Space complexity of a subgraph pattern.

Definition 2.1.3 (Subgraph pattern)**.** Like the database graph, a subgraph pattern is also a directed, labeled and typed graph $\rho = (V_\rho, R_\rho, \lambda_\rho, \mathtt{l}_\rho, \mathtt{t}_\rho, \mathtt{u}_\rho)$.

However, $\mathtt{t}_\rho$ does not map to one single relationship type, but to a non-empty set of allowed relationship types:

$$\mathtt{t}_\rho \ : \ R_\rho \to \mathcal{P}\left(\mathcal{D}_R\right) \setminus \emptyset$$

Moreover, compared to the database graph, a subgraph pattern has one additional component:

$$\mathtt{u}_\rho \subseteq \{\{r, s\} \mid r, s \in R_\rho \wedge r \neq s\}$$

This component will allow us to add uniqueness constraints on pairs of relationship variables (see Definition 2.1.4).

Also, both nodes and relationships of a subgraph pattern are finite, disjoint subsets of a domain of variables $\mathcal{X}$:

1. $V_\rho \subseteq \mathcal{X}$
2. $R_\rho \subseteq \mathcal{X}$
3. $V_\rho \cap R_\rho = \emptyset$

The memory requirements of a subgraph pattern are shown in Table 2.1.

The different components of a subgraph pattern can be best explained by defining the corresponding query result.

Definition 2.1.4 (Query result)**.** The query result res(ρ) of a subgraph pattern $\rho = (V_\rho, R_\rho, \lambda_\rho, \mathtt{l}_\rho, \mathtt{t}_\rho)$ is defined as the set of all subgraphs $\mu \ : \ V_\rho \cup R_\rho \to V \cup R$ that fulfil the following constraints. For all $r, s \in R_\rho$ and $v, w \in V_\rho$:

(1) Node variables are only mapped to nodes and relationship variables only to relationships:

$\mu(r) \in R$ and $\mu(v) \in V$

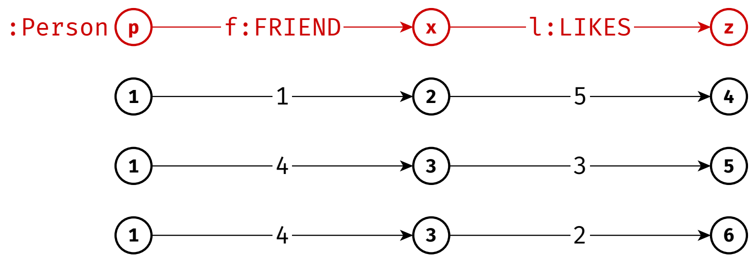


Figure 2.2.: A subgraph pattern and the corresponding query result in the database from Figure 2.1.

(2) Relationships between variables map to relationships in the database:

$\lambda_\rho(r) = (v, w) \Rightarrow \lambda(\mu(r)) = (\mu(v), \mu(w))$

(3) The mapped nodes have at least the labels of the variables:

$\mathtt{l}_\rho(v) \subseteq \mathtt{l}(\mu(v))$

(4) The mapped relationships are in the set of allowed types for the corresponding variable:

$\mathtt{t}(\mu(r)) \in \mathtt{t}_\rho(r)$

(5) If two relationship variables are marked as disjoint, they map to different relationships:

$\{r, s\} \in \mathtt{u}_\rho \Rightarrow \mu(r) \neq \mu(s)$

Consider the subgraph pattern shown in red in Figure 2.2. Node and relationship variables are simply drawn from the alphabet. Each relationship variable is assigned a set of allowed relationship types. In the given example, these sets happen to consist of only one type. Semantically, the example subgraph pattern produces all persons having a friend who likes something. The subgraphs forming the corresponding query result are shown in black. In set notation, the query result is given as:

$$\begin{gathered}\{\{p \mapsto 1_n, f \mapsto 1_r, x \mapsto 2_n, l \mapsto 5_r, z \mapsto 4_n\}\\ \{p \mapsto 1_n, f \mapsto 4_r, x \mapsto 3_n, l \mapsto 3_r, z \mapsto 5_n\}\\ \{p \mapsto 1_n, f \mapsto 4_r, x \mapsto 3_n, l \mapsto 2_r, z \mapsto 6_n\}\}\end{gathered}$$

Let Ω be a query result and $\mu \in \Omega$. It is clear that

$$V_\rho = \{v \mid \mu(v) \in V\} \qquad R_\rho = \{r \mid \mu(r) \in R\}$$

for each subgraph pattern ρ with $\mathrm{res}(\rho) = \Omega$. This means that all subgraph patterns producing a query result do agree on the set of node variables and relationship variables.

Therefore we can write these sets of node variables and relationship variables as functions of Ω:

$$\operatorname{nvar}(\Omega) := \{v \mid \mu(v) \in V \wedge \mu \in \Omega\} \qquad \operatorname{rvar}(\Omega) := \{r \mid \mu(r) \in R \wedge \mu \in \Omega\}$$

For convenience we also define

$$\operatorname{var}(\Omega) := \operatorname{nvar}(\Omega) \cup \operatorname{rvar}(\Omega)$$

2.2. The Algebra

Now that we have a notion of queries on a graph database (subgraph patterns) and have defined the corresponding result sets, we are able to introduce a query algebra. The query algebra is defined on the space of all query results

$$\mathcal{A} := \{\operatorname{res}(\rho) \mid \rho \text{ is a subgraph pattern}\}$$

All functions taking a finite number of query results as inputs and producing another query result as output are possible operators of the algebra:

$$\mathcal{O} := \{o \ : \ \mathcal{A}^k \to \mathcal{A} \mid k \in \mathbb{N}_0\}$$

We will now define a set of operators and show their completeness, i.e., that every query result can be produced by chaining these operators.

2.3. Operators

The GetNodes and Expand operators were first introduced by Hölsch et al. [6].

Definition 2.3.1 (GetNodes operator)**.** The GetNodes operator $\bigcirc_v$ has no inputs and produces the set of all single node subgraphs of the database G.

$$\bigcirc_v := \{\{v \mapsto v'\} \mid v' \in V\}$$

The subgraph pattern producing $\bigcirc_v$ is trivial:

$$\bigcirc_v = \operatorname{res}(\{v\}, \emptyset, \emptyset \to \{v\}^2, \{v \mapsto \emptyset\}, \emptyset \to \mathcal{P}(\mathcal{D}_R) \setminus \emptyset, \emptyset)$$

Therefore GetNodes is a well-defined operator.

Definition 2.3.2 (NodeJoin operator)**.** The NodeJoin operator $\bowtie$ takes two input query results with disjoint relationship variables and merges them at the overlapping node variables.

Let $\mu_1 \in \Omega_1, \mu_2 \in \Omega_2$, $\operatorname{rvar}(\Omega_1) \cap \operatorname{rvar}(\Omega_2) = \emptyset$. Merging the subgraphs μ_1 and μ_2 requires that their node matchings are *compatible* (denoted with $\mu_1 \sim \mu_2$):

$$\mu_1 \sim \mu_2 \quad \Leftrightarrow \quad \forall x \in \operatorname{nvar}(\Omega_1) \cap \operatorname{nvar}(\Omega_2) : \mu_1(x) = \mu_2(x)$$

We define the union of these subgraphs as

$$(\mu_1 \cup \mu_2)(x) := \begin{cases} \mu_1(x), & \text{if } x \in \text{var}(\mu_1), \\ \mu_2(x), & \text{otherwise} \end{cases}$$

Then we define the node join for all query results Ω_1, Ω_2 with $\text{rvar}(\Omega_1) \cap \text{rvar}(\Omega_2) = \emptyset$ as:

$$\Omega_1 \bowtie \Omega_2 := \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1 \wedge \mu_2 \in \Omega_2 \wedge \mu_1 \sim \mu_2\}$$

Proof: NodeJoin is a well-defined operator. Let Ω_1 and Ω_2 be query results and $\rho_1 = (V_{\rho_1}, R_{\rho_1}, \lambda_{\rho_1}, \mathtt{l}_{\rho_1}, \mathtt{t}_{\rho_1}, \mathtt{u}_{\rho_1})$, $\rho_2 = (V_{\rho_2}, R_{\rho_2}, \lambda_{\rho_2}, \mathtt{l}_{\rho_2}, \mathtt{t}_{\rho_2}, \mathtt{u}_{\rho_2})$ be the corresponding subgraph patterns, i.e. $\Omega_1 = \text{res}(\rho_1)$ and $\Omega_2 = \text{res}(\rho_2)$.

The precondition $\text{rvar}(\Omega_1) \cap \text{rvar}(\Omega_2) = \emptyset$ is equivalent to $R_{\rho_1} \cap R_{\rho_2} = \emptyset$.

Let then $\rho = (V_\rho, R_\rho, \lambda_\rho, \mathtt{l}_\rho, \mathtt{t}_\rho, \mathtt{u}_\rho)$ with

$$V_\rho := V_{\rho_1} \cup V_{\rho_2}$$

$$R_\rho := R_{\rho_1} \cup R_{\rho_2}$$

$$\lambda_\rho \ : \ R_\rho \to {V_\rho}^2 \text{ with } \lambda_\rho(r) := \begin{cases} \lambda_{\rho_1}(r), & \text{if } r \in R_{\rho_1} \\ \lambda_{\rho_2}(r), & \text{if } r \in R_{\rho_2} \end{cases}$$

$$\mathtt{l}_\rho(v) := L_1 \cup L_2 \text{ with } L_i := \begin{cases} \mathtt{l}_{\rho_i}(v), & \text{if } v \in V_{\rho_i} \\ \emptyset, & \text{otherwise} \end{cases} \quad \text{for } i \in \{1, 2\}$$

$$\mathtt{t}_\rho(r) := \begin{cases} \mathtt{t}_{\rho_1}(r), & \text{if } r \in R_{\rho_1} \\ \mathtt{t}_{\rho_2}(r), & \text{if } r \in R_{\rho_2} \end{cases}$$

$$\mathtt{u}_\rho := \mathtt{u}_{\rho_1} \cup \mathtt{u}_{\rho_2}$$

We do now have to show that every subgraph in $\Omega_1 \bowtie \Omega_2$ is also in $\text{res}(\rho)$ and vice versa.

($\subseteq$) Take $\mu \in \Omega_1 \bowtie \Omega_2$. Then $\mu = \mu_1 \cup \mu_2$ for some $\mu_1 \in \Omega_1$, $\mu_2 \in \Omega_2, \Omega_1 \sim \Omega_2$.

We show that each constraint of Definition 2.1.4 is fulfilled:

(1) Take $v \in V_\rho$ arbitrary. If $v \in V_{\rho_1}$ then $\mu(v) = \mu_1(v)$ and because $\mu_1 \in \Omega_1$ and Ω_1 is a query result it holds that $\mu_1(v) \in V$. Analogously for $v \in V_{\rho_1}$.

Take $r \in R_\rho$ arbitrary. If $r \in R_{\rho_1}$ then $\mu(r) = \mu_1(r)$ and because $\mu_1 \in \Omega_1$ and Ω_1 is a query result it holds that $\mu_1(r) \in R$. Analogously for $r \in R_{\rho_2}$.

(2) Assume $\lambda_\rho(r) = (v, w)$ for some $r \in R_{\rho_1}$ and $v, w \in V_\rho$.

Then $\lambda_\rho(r) = \lambda_{\rho_1}(r) = (v, w)$. Because $\mu_1 \in \text{res}(\rho_1)$ it follows by constraint (2) that $\lambda(\mu_1(r)) = (\mu_1(v), \mu_1(w))$.

Because Ω_1 and Ω_2 are compatible it holds that $\mu_1(x) = \mu(x)$ for all $x \in V_{\rho_1} \cup R_{\rho_1}$. Therefore $\lambda(\mu(r)) = (\mu(v), \mu(w))$.

Analogously for $r \in R_{\rho_2}$.

(3) Take $v \in V_\rho$ arbitrary.

If $v \in V_{\rho_1} \setminus V_{\rho_2}$ then $\mathtt{l}_\rho(v) = \mathtt{l}_{\rho_1}(v) \subseteq \mathtt{l}(\mu_1(v)) = \mathtt{l}(\mu(v))$.

Analogously for $v \in V_{\rho_2} \setminus V_{\rho_1}$.

If $v \in V_{\rho_2} \cap V_{\rho_1}$ then $\mathtt{l}_\rho(v) = \mathtt{l}_{\rho_1}(v) \cup \mathtt{l}_{\rho_2}(v) \subseteq \mathtt{l}(\mu_1(v)) \cup \mathtt{l}(\mu_2(v)) = \mathtt{l}(\mu(v))$.

(4) Take $r \in R_\rho$ arbitrary.

If $r \in R_{\rho_1}$ then $\mathtt{t}_\rho(r) = \mathtt{t}_{\rho_1}(r) \ni \mathtt{t}(\mu_1(r)) = \mathtt{t}(\mu(r))$.

Analogously for $r \in R_{\rho_2}$.

(5) Take $\{r, s\}\mathtt{u}$ arbitrary.

If $\{r, s\} \in \mathtt{u}_{\rho_1}$, then $\mu(r) = \mu_1(r) \neq \mu_1(s) = \mu(s)$ because $\mu_1 \in \text{res}(\rho_1)$.

Analogously for $\{r, s\} \in \mathtt{u}_{\rho_2}$.

Therefore $\mu \in \text{res}(\rho)$.

($\supseteq$) Take $\mu \in \text{res}(\rho)$.

Let $\mu_1 : V_{\rho_1} \cup R_{\rho_1} \to V \cup R$ with $\mu_1(x) := \mu(x)$ and $\mu_2 : V_{\rho_2} \cup R_{\rho_2} \to V \cup R$ with $\mu_2(x) := \mu(x)$.

Then $\mu_1 \in \text{res}(\rho_1) = \Omega_1$ and $\mu_2 \in \text{res}(\rho_2) = \Omega_2$.

Also $\mu_1 \sim \mu_2$.

Therefore $\mu \in \Omega_1 \bowtie \Omega_2$.

We have shown that, for any input results $\Omega_1 = \text{res}(\rho_1)$, $\Omega_2 = \text{res}(\rho_2)$, there is a subgraph pattern ρ with $\text{res}(\rho) = \Omega_1 \bowtie \Omega_2$.

Therefore NodeJoin is a well-defined query operator mapping from $\mathcal{A}^2$ to $\mathcal{A}$. $\square$

Example 2.3.1. It holds that

$$\begin{aligned} &\text{res}(\texttt{:Person}\,\textcircled{p} \xrightarrow{\texttt{f:FRIEND}} \textcircled{x}) \bowtie \text{res}(\textcircled{x} \xrightarrow{\texttt{l:LIKES}} \textcircled{z}) \\ &= \text{res}(\texttt{:Person}\,\textcircled{p} \xrightarrow{\texttt{f:FRIEND}} \textcircled{x} \xrightarrow{\texttt{l:LIKES}} \textcircled{z}) \end{aligned}$$

Definition 2.3.3 (Traverse operator)**.** The Traverse operator $\tau_{v \overset{r:T}{\alpha} w}(\Omega)$ takes a query result Ω as input and expands it by adding new relationships between nodes. The precondition is that $v, w \in \text{nvar}(\Omega)$. The direction of the relationships is specified by $\alpha \in \{\leftarrow, \rightarrow\}$ and the allowed relationship types by $T \subseteq \mathcal{D}_R$.

For $\lambda(s) = (p, q)$ we set $\lambda_\rightarrow(s) = (p, q)$ and $\lambda_\leftarrow(s) = (q, p)$.

Then we define

$$\tau_{v \overset{r:T}{\alpha} w}(\Omega) := \{\mu \cup \{r \mapsto r'\} \mid \mu \in \Omega \wedge \lambda_\alpha(r') = (\mu(v), \mu(w)) \wedge t(r') \in T\}$$

Proof: Traverse is a well-defined operator. Let Ω be a query result and $\rho' = (V_{\rho'}, R_{\rho'}, \lambda_{\rho'}, \mathtt{l}_{\rho'}, \mathtt{t}_{\rho'}, \mathtt{u}_{\rho'})$ the corresponding subgraph pattern.

Let then $\rho = (V_\rho, R_\rho, \lambda_\rho, \mathtt{l}_\rho, \mathtt{t}_\rho, \mathtt{u}_\rho)$ with

$$V_\rho := V_{\rho'}$$

$$R_\rho := R_{\rho'} \cup \{r\}$$

$$\lambda_\rho(x) := \begin{cases} (v,w) \text{ if } x = r \ \wedge \ \alpha = \rightarrow \\ (w,v) \text{ if } x = r \ \wedge \ \alpha = \leftarrow \\ \lambda_{\rho'}(x), \text{ otherwise} \end{cases}$$

$$\mathtt{l}_\rho := \mathtt{l}_{\rho'}$$

$$\mathtt{t}_\rho(x) := \begin{cases} T, \text{ if } x = r \\ \mathtt{t}_{\rho'}(x), \text{ otherwise} \end{cases}$$

$$\mathtt{u}_\rho := \mathtt{u}_{\rho'}$$

Again we have to show that every subgraph in $\tau_{v \overset{r:T}{\alpha} w}(\Omega)$ is also in $\text{res}(\rho)$ and vice versa.

($\subseteq$) Take $\mu \in \tau_{v \overset{r:T}{\alpha} w}(\Omega)$ arbitrary. Then $\mu = \mu' \cup \{r \mapsto r'\}$ for some $\mu' \in \Omega$, $r' \in R$, such that $\lambda_\alpha(r') = (\mu'(v), \mu'(w'))$.

We show again that each constraint of Definition 2.1.4 is fulfilled:

(1) $\mu(r) = r' \in R$ by definition. For all $v \in V_\rho, s \in R_\rho \setminus \{r\}$: $\mu(v) = \mu'(v) \in V$ and $\mu(s) = \mu'(s) \in R$ because $\mu' \in \Omega$ and Ω is a query result.

(2) If $\alpha = \rightarrow$ then $\lambda_\rho(r) = (v,w)$ and $\lambda(\mu(r)) = \lambda(r') = \lambda_\rightarrow(r') = (v',w') = (\mu(v), \mu(w))$.

If $\alpha = \leftarrow$ then $\lambda_\rho(r) = (w,v)$ and $\lambda(\mu(r)) = \lambda(r') = \lambda_\rightarrow(r') = (w',v') = (\mu(w), \mu(v))$.

For $x \in R_\rho, x \neq r$ it holds that if $\lambda_\rho(x) = \lambda_{\rho'}(x) = (p,q)$ then $\lambda(\mu(x)) = \lambda(\mu'(x)) = (\mu'(p), \mu'(q)) = (\mu(p), \mu(q))$ because $\mu' \in \Omega$ and Ω is a query result.

(3) Take $v \in V_\rho$ arbitrary. $\mathtt{l}_\rho(v) = \mathtt{l}_{\rho'}(v) \subseteq \mathtt{l}(\mu'(v)) = \mathtt{l}(\mu(v))$.

(4) $\mathtt{t}_\rho(r) = T \ni t(r') = t(\mu(r))$ by definition.

For $s \in R_\rho, s \neq r$ it holds that $\mathtt{t}_\rho(s) = \mathtt{t}_{\rho'}(s) \ni \mathtt{t}(\mu'(s)) = \mathtt{t}(\mu(s))$.

(5) Take $\{s,t\} \in \mathtt{u}_\rho = \mathtt{u}_{\rho'}$ arbitrary. Then $s,t \in R_{\rho'}$ and therefore $\mu(s) = \mu'(s)$ and $\mu(t) = \mu'(t)$.

Because $\mu' \in \text{res}(\rho')$ it follows that $\mu(s) = \mu'(s) \neq \mu'(t) = \mu(t)$.

As a result, $\mu \in \text{res}(\rho)$.

($\supseteq$) Take $\mu \in \Omega = \text{res}(\rho)$ arbitrary. Let $\mu' : V_{\rho'} \cup R_{\rho'} \to V \cup R$ with $\mu'(x) := \mu(x)$. Then $\mu' \in \text{res}(\rho') = \Omega$.

If $\alpha = \rightarrow$ then $\lambda_\rho(r) = (v,w)$ and because of constraint (2) $\lambda_\alpha(\mu(r)) = \lambda(\mu(r)) = (\mu(v), \mu(w))$.

Analogously for $\alpha = \leftarrow$.

Finally, $\mathtt{t}(r') = \mathtt{t}(\mu(r)) \in \mathtt{t}_\rho(r) = T$.

Therefore, $\mu \in \tau_{v \overset{r:T}{\alpha} w}(\Omega)$.

It follows that $\tau_{v \overset{r:T}{\alpha} w}(\Omega) = \mathrm{res}(\rho)$. We conclude that Traverse is a well-defined operator. $\square$

Definition 2.3.4 (Expand operator)**.** The Expand operator $\varepsilon_{v \overset{r:T}{\alpha} w}(\Omega)$ takes a query result Ω as input and expands it by adding new nodes using relationships. The preconditions are that $v \in \mathrm{nvar}(\Omega)$ and $w \notin \mathrm{nvar}(\Omega)$ The direction of the relationships is specified by $\alpha \in \{\leftarrow, \rightarrow\}$ and the allowed relationship types by $T \subseteq \mathcal{D}_R$.

For $\lambda(r) = (v, w)$ we set $\lambda_\rightarrow(r) = (v, w)$ and $\lambda_\leftarrow(r) = (w, v)$.

Then we define

$$\varepsilon_{v \overset{r:T}{\alpha} w}(\Omega) := \{\mu \cup \{r \mapsto r', w \mapsto w'\} \mid \mu \in \Omega \wedge \lambda_\alpha(r') = (\mu(v), \mu(w)) \wedge \mathtt{t}(r') \in T\}$$

Proof: Expand is a well-defined operator. By definition it holds that

$$\begin{aligned}
\varepsilon_{v \overset{r:T}{\alpha} w}(\Omega) &= \{\mu \cup \{r \mapsto r'\} \cup \{w \mapsto w'\} \mid \mu \in \Omega \wedge \lambda_\alpha(r') = (\mu(v), \mu(w)) \wedge \mathtt{t}(r') \in T\} \\
&= \{\mu_1 \cup \mu_2 \cup \{r \mapsto r'\} \mid \mu_1 \in \Omega \wedge \mu_2 \in \bigcirc_w \wedge \lambda_\alpha(r') = (\mu_1(v), \mu_2(w)) \wedge \mathtt{t}(r') \in T\} \\
&= \{\mu \cup \{r \mapsto r'\} \mid \mu \in \Omega \bowtie \bigcirc_w \wedge \lambda_\alpha(r') = (\mu(v), \mu(w)) \wedge \mathtt{t}(r') \in T\} \\
&= \tau_{v \overset{r:T}{\alpha} w}(\Omega \bowtie \bigcirc_w)
\end{aligned} \tag{2.1}$$

Because the operators GetNodes, Join and Traverse are well-defined, the Expand operator is also well-defined. $\square$

Remark 2. Because the Expand operator can be defined in terms of GetNodes, Join and Traverse, it is not needed from a logical perspective.

However, some graph databases (e.g. Neo4j) provide special physical operators implementing the Expand operator in an efficient manner. Thus there is an interest in having this operator available as a shorthand for the combination of canonical operators shown in Equation 2.1.

Example 2.3.2. It holds that

$$\varepsilon_{t \xleftarrow{f:\{\mathrm{FRIEND}\}} p}(\mathrm{res}(\textcircled{t}\,\mathtt{:Teacher})) = \mathrm{res}(\textcircled{p}\text{—}\mathtt{f:FRIEND}\rightarrow\textcircled{t}\,\mathtt{:Teacher})$$

Definition 2.3.5 (LabelSelection operator)**.** The LabelSelection operator $\sigma_{v:l}(\Omega)$ takes a query result Ω as input and returns only those subgraphs where the node matched by v has the label l.

It is defined as

$$\sigma_{v:l}(\Omega) := \{\mu \in \Omega \mid l \in \mathtt{l}(\mu(v))\}$$

Proof: LabelSelection is a well-defined operator. Let Ω be a query result and $\rho' = (V_{\rho'}, R_{\rho'}, \lambda_{\rho'}, \mathtt{l}_{\rho'}, \mathtt{t}_{\rho'}, \mathtt{u}_{\rho'})$ the corresponding subgraph pattern.

Let then $\rho = (V_\rho, R_\rho, \lambda_\rho, \mathtt{l}_\rho, \mathtt{t}_\rho, \mathtt{u}_\rho)$ with

$$V_\rho := V_{\rho'}$$

$$R_\rho := R_{\rho'}$$

$$\lambda_\rho := \lambda_{\rho'}$$

$$\mathtt{l}_\rho(x) := \begin{cases} \mathtt{l}_{\rho'} \cup \{l\}, & \text{if } x = v \\ \mathtt{l}\rho'(x), & \text{otherwise} \end{cases}$$

$$\mathtt{t}_\rho := \mathtt{t}_{\rho'}$$

$$\mathtt{u}_\rho := \mathtt{u}_{\rho'}$$

Once more we have to show that every subgraph in $\sigma_{v:l}(\Omega)$ is also in $\mathrm{res}(\rho)$ and vice versa.

($\subseteq$) Take $\mu \in \sigma_{v:l}(\Omega)$ arbitrary. Then $\mu \in \Omega$ with $l \in \mathtt{l}(\mu(v))$.

We show again that each constraint of Definition 2.1.4 is fulfilled (but with less detail, to avoid repetition):

(1) Fulfilled, because $\mu \in \Omega$ and Ω is a query result.

(2) Fulfilled, because $\lambda_\rho = \lambda_{\rho'}$ and $\mu \in \mathrm{res}(\rho')$.

(3) First of all, by definition $\{l\} \subseteq \mathtt{l}(\mu(v))$. Also $\mathtt{l}_{\rho'}(v) \subseteq \mathtt{l}(\mu(v))$, because $\mu \in \Omega$ and Ω is a query result. Therefore, $\mathtt{l}_\rho(v) = \{l\} \cup \mathtt{l}_{\rho'}(v) \subseteq \mathtt{l}(\mu(v))$

(4) Fulfilled, because $\mathtt{t}_\rho = \mathtt{t}_{\rho'}$ and $\mu \in \mathrm{res}(\rho')$.

(5) Fulfilled, because and $\mathtt{u}_\rho = \mathtt{u}_{\rho'}$ and $\mu \in \mathrm{res}(\rho')$.

As a result, $\mu \in \mathrm{res}(\rho)$.

($\supseteq$) Take $\mu \in \mathrm{res}(\rho)$ arbitrary. Then $\mu \in \mathrm{res}(\rho') = \Omega$, because the subgraph pattern ρ has a superset of the constraints of ρ'.

Also $\mathtt{l}_\rho(v) = \mathtt{l}_{\rho'}(v) \cup \{l\} \subseteq \mathtt{l}(\mu(v))$ which implies that $l \in \mathtt{l}(\mu(v))$.

Therefore, $\mu \in \sigma_{v:l}(\Omega)$.

We have proven that $\sigma_{v:l}(\Omega) = \mathrm{res}(\rho)$ and conclude that LabelSelection is a well-defined operator. □

Definition 2.3.6 (DistinctSelection operator)**.** The DistinctSelection operator $\sigma_{r \neq s}(\Omega)$ takes a query result Ω as input and returns only those subgraphs where the relationships matched by the variables r and s are different.

It is defined as

$$\sigma_{r \neq s}(\Omega) := \{\mu \in \Omega \mid \mu(r) \neq \mu(s)\}$$

Proof: DistinctSelection is a well-defined operator. Let Ω be a query result and $\rho' = (V_{\rho'}, R_{\rho'}, \lambda_{\rho'}, \mathtt{l}_{\rho'}, \mathtt{t}_{\rho'}, \mathtt{u}_{\rho'})$ the corresponding subgraph pattern.

Let then $\rho = (V_\rho, R_\rho, \lambda_\rho, \mathtt{l}_\rho, \mathtt{t}_\rho, \mathtt{u}_\rho)$ with

$$V_\rho := V_{\rho'}$$

$$R_\rho := R_{\rho'}$$

$$\lambda_\rho := \lambda_{\rho'}$$

$$\mathtt{l}_\rho := \mathtt{l}_{\rho'}$$

$$\mathtt{t}_\rho := \mathtt{t}_{\rho'}$$

$$\mathtt{u}_\rho := \mathtt{u}_{\rho'} \cup \{\{r, s\}\}$$

Once more we have to show that every subgraph in $\sigma_{r\neq s}(\Omega)$ is also in $\mathrm{res}(\rho)$ and vice versa.

($\subseteq$) Take $\mu \in \sigma_{r\neq s}(\Omega)$ arbitrary. Then $\mu \in \Omega$ with $\mu(r) \neq \mu(s)$.

We show again that each constraint of Definition 2.1.4 is fulfilled:

(1) Fulfilled, because $\mu \in \Omega$ and Ω is a query result.

(2) Fulfilled, because $\lambda_\rho = \lambda_{\rho'}$ and $\mu \in \mathrm{res}(\rho')$.

(3) Fulfilled, because $\mathtt{l}_\rho = \mathtt{l}_{\rho'}$ and $\mu \in \mathrm{res}(\rho')$.

(4) Fulfilled, because $\mathtt{t}_\rho = \mathtt{t}_{\rho'}$ and $\mu \in \mathrm{res}(\rho')$.

(5) By definition it holds that $\{r, s\} \in \mathtt{u}_\rho$ and $\mu(r) \neq \mu(s)$.

Take now $\{t, u\} \in \mathtt{u}_\rho$ with $\{t, u\} \neq \{r, s\}$. Then $\{t, u\} \in \mathtt{u}_{\rho'}$ and because $\mu \in \mathrm{res}(\rho')$ it follows that $\mu(t) \neq \mu(u)$.

As a result, $\mu \in \mathrm{res}(\rho)$.

($\supseteq$) Take $\mu \in \mathrm{res}(\rho)$ arbitrary. Then $\mu \in \mathrm{res}(\rho') = \Omega$, because the subgraph pattern ρ has a superset of the constraints of ρ'. Because $\{r, s\} \in \mathtt{u}_\rho$, we have $\mu(r) \neq \mu(s)$.

Therefore, $\mu \in \sigma_{r\neq s}(\Omega)$.

We have proven that $\sigma_{r\neq s}(\Omega) = \mathrm{res}(\rho)$ and conclude that DistinctSelection is a well-defined operator. $\square$

2.4. Completeness

A set of operators is *complete*, iff it is possible to produce the whole result space of the corresponding algebra by chaining these operators.

We are interested in the set

$$\mathcal{O}_c := \{\bigcirc_v, \bowtie, \tau_{v \overset{r:T}{\alpha} w}, \sigma_{v:l}, \sigma_{r\neq s}\} \subset \mathcal{O}$$

Proof: $\mathcal{O}_c$ is a complete set of operators w.r.t. $\mathcal{A}$. Let $\rho = (V_\rho, R_\rho, \lambda_\rho, \mathtt{l}_\rho, \mathtt{t}_\rho, \mathtt{u}_\rho)$ be a subgraph pattern. We have to show that there is a combination of query operators from $\mathcal{O}_c$ producing $\text{res}(\rho)$.

First of all, because V_ρ, R_ρ are finite, we can enumerate them in a sequence:

$$V_\rho = \{v_1, v_2, \ldots, v_{|V_\rho|}\}, \quad R_\rho = \{r_1, r_2, \ldots, r_{|R_\rho|}\}$$

The same holds for all labels of a node variable $v_i \in V_\rho$

$$\mathtt{l}_\rho(v_i) = \{l_{i,1}, l_{i,2}, \ldots, l_{i,|\mathtt{l}_\rho(v_i)|}\}$$

and for all pairs of relationships in $\mathtt{u}_\rho$

$$\mathtt{u}_\rho = \{\{r_{1,1}, r_{1,2}\}, \ldots, \{r_{|\mathtt{u}_\rho|,1}, r_{|\mathtt{u}_\rho|,2}\}\}$$

We can now reuse some work done in the previous proofs, where we showed that the query operators are well-defined. For each operator we defined an output subgraph pattern in terms of the input subgraph patterns. The idea for the completeness proof is the following: We build a chain of query operators which we suppose to equal $\text{res}(\rho)$. Then we use the mentioned pattern definitions to find a pattern ρ' that produces the same query result as this chain. We will see that $\rho' = \rho$ and therefore $\text{res}(\rho') = \text{res}(\rho)$.

Let us start to build the chain of operators.

1. We begin with the following query result:

   $$\begin{aligned} J_1 &:= \bigcirc_{v_1} \\ J_i &:= \bigcirc_{v_i} \bowtie J_{i-1} \quad \text{for } i \in \{2, \ldots, |V_\rho|\} \end{aligned}$$

   In Definition 2.3.1 we found that, $J_1 = \bigcirc_{v_1}$ can be produced by a pattern that has only one node variable, no relationships and no type or label constraints. Using Definition 2.3.2 we observe that, given a pattern ρ_{i-1} producing J_{i-1}, a pattern ρ_i producing J_i can be obtained by simply adding the variable v_i to the set of nodes of ρ_{i-1}. All other components of ρ_i equal those of ρ_{i-1}.

   Consequently, the following pattern ρ_J having all node variables V_ρ but no relationships, no type constraints and no label constraints will produce $J_{|V_\rho|}$:

   $\rho_J = (V_\rho, \emptyset, \emptyset \to {V_\rho}^2, V_\rho \to \emptyset, \emptyset \mapsto \mathcal{P}(\mathcal{D}_R) \setminus \emptyset, \emptyset)$.

2. In the next step, we match relationships between the nodes in $J_{|V_\rho|}$.

   For a tuple $t = (t_1, t_2, \ldots, t_n)$ we denote the projection on the i'th component with $\pi_i(t) := t_i$.

   We construct a sequence of TRAVERSE operators matching all relationships that exist in ρ:

   $$\begin{aligned} R_0 &:= J_{|V_\rho|} \\ R_i &:= \tau_{\pi_1(\lambda_\rho(r_i)) \xrightarrow{r_i : \mathtt{t}_\rho(r_i)} \pi_2(\lambda_\rho(r_i))}(R_{i-1}) \end{aligned}$$

Incrementing the parameter from $i-1$ to i means to append a TRAVERSE operator for relationship r_i to R_{i-1}. According to the proof that TRAVERSE is well-defined (cf. Definition 2.3.3), this is equivalent to adding the relationship r_i to the corresponding subgraph pattern, together with the type constraint.

It is easy to see that, $R_{|R_\rho|} = \text{res}(\rho_R)$ with $\rho_R = (V_\rho, R_\rho, \lambda_\rho, V_\rho \to \{\emptyset\}, \mathtt{t}_\rho, \mathtt{u}_\rho)$. This pattern already equals ρ in all components except the label function and the set of distinct relationship pairs.

3. In the second last step, we shrink the set of subgraphs by adding label constraints.

   We define

$$\begin{aligned} L_{0,0} &:= R_{|R_\rho|} \\ L_{i,0} &:= L_{i-1,|\mathtt{l}_\rho(v_i)|} \quad \text{for } i \in \{1, \ldots, |V_\rho|\} \\ L_{i,j} &:= \sigma_{v_i:l_{i,j}}(L_{i,j-1}) \quad \text{for } i \in \{1, \ldots, |V_\rho|\}, j \in \{1, \ldots, |\mathtt{l}(v_i)|\} \end{aligned}$$

   We are interested in $L := L_{|V_\rho|,\left|\mathtt{l}_\rho(v_{|V_\rho|})\right|}$. As in the previous two steps, we iteratively construct a subgraph pattern which produces the same result, using the recursive definition from the proof that the LABELSELECTION operator is well-defined (cf. Definition 2.3.5).

   Going from $L_{i,j}$ to $L_{i,j-1}$ means to append a LABELSELECTION for label $l_{i,j}$ on the node variable v_i. According to the proof that LABELSELECTION is well-defined (cf. Definition 2.3.5), this is equivalent to adding a new label to $\mathtt{l}(v_i)$ in the corresponding subgraph pattern. All other components of the pattern do not change.

   We see that L contains exactly one LABELSELECTION for each label constraint stored in $\mathtt{l}_\rho$. Therefore, going from the previous pattern ρ_R to a pattern ρ_L with $\text{res}(\rho_L) = L$, means adding all of the label constraints from $\mathtt{l}_\rho$ to ρ_R. Because there are no label constraints in ρ_R, the label function in ρ_L simply equals $\mathtt{l}_\rho$.

   Thus the generated pattern is

$$\rho_L = (V_\rho, R_\rho, \lambda_\rho, \mathtt{l}_\rho, \mathtt{t}_\rho, \emptyset)$$

4. In the final step, we shrink the set of subgraphs by introducing constraints ensuring relationship uniqueness.

   We define

$$\begin{aligned} U_0 &:= L \\ U_i &:= \sigma_{r_{i,1} \neq r_{i,2}}(U_{i-1}) \quad \text{for } i \in \{1, \ldots, |\mathtt{u}_\rho|\} \end{aligned}$$

   We are interested in $U := U_{|\mathtt{u}_\rho|}$. We proceed as in the previous steps.

   Going from U_i to U_{i-1} means to append a DISTINCTSELECTION operator for the relationships $r_{i,1}, r_{i,2}$. According to the proof that DISTINCTSELECTION is well-defined (cf. Definition 2.3.6), this is equivalent to adding the pair $\{r_{i,1}, r_{i,2}\}$ to the set of distinct relationship pairs in the corresponding subgraph pattern. All other components of the pattern do not change.

We see that U contains exactly one DISTINCTSELECTION operator for each pair of relationship variables stored in $\mathtt{u}_\rho$. Therefore, going from the previous pattern ρ_L to ρ_U with $\text{res}(\rho_U) = U$, means adding all of the pairs from $\mathtt{u}_\rho$ to ρ_L. Because the set of distinct relationship pairs of ρ_L is empty, the set of distinct relationship pairs of ρ_U simply equals $\mathtt{u}_\rho$.

Thus the generated pattern is

$$\rho_U = (V_\rho, R_\rho, \lambda_\rho, \mathtt{l}_\rho, \mathtt{t}_\rho, \mathtt{u}_\rho) = \rho$$

It follows that

$$U = \text{res}(\rho_U) = \text{res}(\rho) \ .$$

□

3. Assumptions About the Distribution of Labels

In this chapter, we introduce three possible assumptions about the distribution of node labels and show that none of them is likely to be valid for the whole database. To address this problem, we propose two data structures to decide which assumption to use for a particular set of labels.

3.1. The Problem of Global Assumptions

In the relational algebra, a result of tuples can be filtered according to a predicate p using the selection operator σ_p. Estimating the result cardinality of the selection is equivalent to estimating the probability of the predicate.

To estimate the probability of a predicate that references multiple attributes, it is essential to know how the value combinations of these attributes are distributed. A simple approach is to assume probabilistic independence of the attributes. The probability of a combination of attribute values is then simply the product of the individual value probabilities, which are stored in the system catalog.

However, the attribute value independence assumption is often violated in real data sets (e.g. when there are functional dependencies). Consequently, research has been done to find better ways of estimating the result cardinality of multi-attribute selections [9].

In our graph algebra, nodes can be filtered using the NodeLabelSelection operator. Suppose the graph DBMS maintains a relation R with schema $< n, l_1, \ldots, l_i >$, where n is a node and $l_1, \ldots, l_i$ are Booleans assigning a set of labels to this node. Estimating the result cardinality of several consecutive label selections is then equivalent to estimating the result cardinality of a multi- attribute selection on the $l_1, \ldots, l_i$ attributes of this table. Consequently, label selections in a graph database can be viewed as a special case of multi-attribute selections in a relational database.

Take for instance two labels called "Person" and "Student". Semantically, every student is a person. Therefore $\sigma_{v:\text{Student}}(\sigma_{v:\text{Person}}(\bigcirc_v)) = \sigma_{v:\text{Student}}(\bigcirc_v)$. We say that "Student" is a sublabel of "Person" (see Definition 3.1.1). Obviously, if the database knows about this sublabel relationship, it will make a far better estimation of the result cardinality than if it assumes e.g., independence of students and persons.

Remark 3. Relationship types resemble node labels, because they are also a way of assigning a group membership to a set of entities, namely relationships. However, every relationship is exactly of one type. As a result, there is no uncertainty about the distribution of relationship types, they are disjoint (unlike node labels).

Nevertheless, the popular graph database Neo4j always assumes independence of labels. In this thesis, we want to demonstrate that cardinality estimates can be significantly improved by incorporating more information about the node label distribution. In particular, we consider the cases that labels are *disjoint* or in a *sublabel* relation.

Definition 3.1.1 (Sublabels)**.** A label l' is a sublabel of l, iff $\sigma_{v:l'}(\bigcirc_v) \subseteq \sigma_{v:l}(\bigcirc_v)$. Equivalently, we call l a superlabel of l'. We denote this relation by $l' \subseteq l$.

Let us now formalize possible assumptions on the node label distribution.

Take a query result Ω. Ω can be interpreted as a finite probability space, such that each subgraph $\mu \in \Omega$ is an elementary event with probability $\mathrm{P}[\Omega](\{\mu\}) = \frac{1}{|\Omega|}$. Then a label selection $\sigma_{v:l}(\Omega)$ is an event in Ω. We set $v{:}l := \sigma_{v:l}(\Omega)$ for readability. The probability of drawing a subgraph from Ω where the node matched by the variable v has label l is given as

$$\mathrm{P}[\Omega](v{:}l) = \frac{|\sigma_{v:l}(\Omega)|}{|\Omega|} \tag{3.1}$$

The problem is to estimate the probability

$$\mathrm{P}[\Omega]\left(\textstyle\bigcap_{l\in L} v{:}l\right)$$

for a set $L \subseteq \mathcal{D}_L$ of node labels. We write $v{:}L$ as a short-hand for $\bigcap_{l\in L} v{:}l$.

We already suggested that, making a global assumption about the relationship of all pairs of labels $l_1, l_2 \in L$ is error-prone. To illustrate this fact, we look at three different global assumptions and the corresponding estimation formulas.

1. Suppose the labels form a subset chain, i.e., for all $l_1, l_2 \in L$ it holds either $l_1 \subseteq l_2$ or $l_2 \subseteq l_1$. Then
$$\mathrm{P}[\Omega](v{:}L) = \min_{l\in L} \mathrm{P}[\Omega](v{:}l)$$
2. Suppose the labels are pair-wise independent, i.e., for all $l_1, l_2 \in L$ it holds that $\mathrm{P}[\Omega](v{:}l_1 \cap v{:}l_2) = \mathrm{P}[\Omega](v{:}l_1) \cdot \mathrm{P}[\Omega](v{:}l_2)$. Then
$$\mathrm{P}[\Omega](v{:}L) = \prod_{l\in L} \mathrm{P}[\Omega](v{:}l)$$
3. Suppose there are two labels $l_1, l_2 \in L$ that are disjoint, i.e., $\mathrm{P}[\Omega](v{:}l_1 \cap v{:}l_2) = 0$. Then
$$\mathrm{P}[\Omega](v{:}L) = 0$$

Without any knowledge about the database graph, it is impossible to tell which of these assumptions should be preferred. In particular, we have no reason to choose the independence assumption, other than assuming that our database will only be used for datasets where node labels are independent.

Moreover, none of these assumptions is likely to be valid for all subsets of labels L. From a user perspective, it seems very probable that there will be labels that are in a sublabel relation (recall the example on students and persons above) and at the same time labels that are disjoint (consider for instance two labels "Male" and "Female"). Hence, we need means to decide which assumption to use for a given set of labels.

Remark 4. Note that labels being disjoint or in a sublabel relation does not imply that there is a functional dependency between the attributes representing these labels in the relation R described above:

If two labels l_1, l_2 are disjoint, there are no nodes having both of these labels, i.e., there are no tuples $t \in R$ with $t[l_1] = \texttt{true} = t[l_2]$. Similarly, if $l_1 \subseteq l_2$, there are no nodes having l_1 but not l_2, i.e., there are no tuples $t \in R$ with $t[l_1] = \texttt{true}$ and $t[l_2] = \texttt{false}$. However, in both cases it is possible that two tuples $t_1, t_2 \in R$ with $t_1[l_1] = \texttt{false}$, $t_1[l_2] = \texttt{false}$, $t_2[l_1] = \texttt{false}$ and $t_2[l_2] = \texttt{true}$.

Consequently, techniques for functional dependency discovery are not suitable for the discovery of sublabels or disjoint labels.

3.2. Storing More Information

We will now introduce two data structures that allow us to decide which assumption to use for a given set of labels.

3.2.1. Label Partition

First of all, we divide the set of labels $\mathcal{D}_L$ in the database graph into subsets of *overlapping* labels, that form a partition D of this set. For a given label l, we denote the member set of D containing l as $D(l)$. If two labels are in different member sets of D, we will assume they are disjoint. We refer to such a partition as a *label partition*.

A label partition may be provided *a priori* by the user. It can then be understood as a contract: The user does not intend to assign two labels from different member sets of the partition to the same node.

A label partition can also be built *a posteriori* by the DBMS, e.g. using a clustering algorithm. The distance measure for two labels l_1, l_2 is the Jaccard distance between their sets of nodes:

$$\begin{aligned} d(l_1, l_2) &= d_J(\sigma_{v:l_1}(\bigcirc_v), \sigma_{v:l_2}(\bigcirc_v)) \\ &= 1 - \frac{|\sigma_{v:l_1}(\sigma_{v:l_2}(\bigcirc_v))|}{|\sigma_{v:l_1}(\bigcirc_v) \cup \sigma_{v:l_2}(\bigcirc_v)|} \end{aligned} \tag{3.2}$$

The clusters produced by an algorithm using this distance measure represent the wanted partition. Obviously, the result strongly depends on the chosen algorithm and its parameters.

To better understand the intention of the label partition, consider the following example. Suppose we have a database about movies. Figure 3.1 shows the distribution of the different labels among the nodes of this database as a Venn diagram.

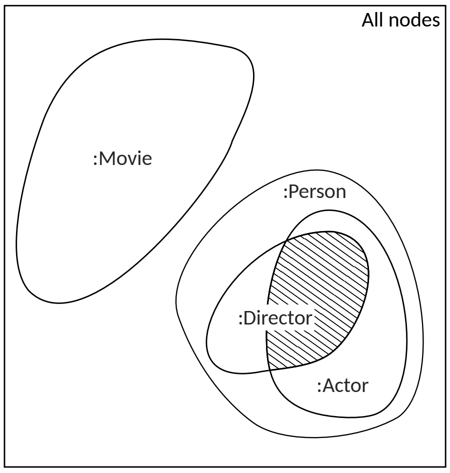


Figure 3.1.: Distribution of node labels in a example database.

From this diagram, we can conclude that movies and persons are disjoint. We also see that, persons, actors and directors do strongly overlap. Consequently, the following label partition might be a good choice:

$$D := \{\{\text{Movie}\}, \{\text{Person}, \text{Actor}, \text{Director}\}\}$$

We highlight one option for a partition, which we call the *strict partition.* First, we define a relation for overlapping labels:

$$\begin{aligned} &\text{overlap} \subseteq \mathcal{D}_L^2 \\ &l_1 \text{ overlap } l_2 \ :\Leftrightarrow\ \sigma_{v:l_1}(\sigma_{v:l_2}(\bigcirc_v)) \neq \emptyset \end{aligned} \tag{3.3}$$

overlap is reflexive and symmetric and the transitive closure overlap$^+$ is thus an equivalence relation.

The strict partition $\widetilde{D}$ is simply the set of equivalence classes of overlap$^+$ (also called the quotient set):

$$\begin{aligned} \widetilde{D} &:= \mathcal{D}_L/\text{overlap}^+ \\ &= \{\{l' \in \mathcal{D}_L \mid l \text{ overlap}^+ \ l'\} \mid l \in \mathcal{D}_L\} \end{aligned} \tag{3.4}$$

We observe that labels from different equivalence classes are disjoint: Take two labels l_1, l_2 from different equivalence classes $O_1, O_2 \in \widetilde{D}$. Then $(l_1, l_2) \notin \text{overlap}^+$. This implies $(l_1, l_2) \notin$ overlap and therefore $\sigma_{v:l_1}(\sigma_{v:l_2}(\bigcirc_v)) = \emptyset$. Note that the inverse is not necessarily true, there may be two labels in the same equivalence class which are disjoint (this is a potential weakness of the strict partition, see the remark below).

As a result, if $L \subseteq \mathcal{D}_L$ contains two labels from different members of $\widetilde{D}$ then $\mathrm{P}[\Omega](v{:}L) = 0$ for any query result Ω.

Remark 5. The strict partition can also be obtained using a single-linkage hierarchical clustering that succesively merges all clusters of labels having a cluster distance smaller than 1 (using the Jaccard distance measure defined in Equation 3.2). Once the single-linkage algorithm has finished, two labels from different clusters will have distance 1, which means that their node sets are disjoint.

Because the single-linkage algorithm tends to build long chains, it can happen that many pairs of disjoint node labels end up in the same cluster. This can be avoided by a different linkage criterion (e.g. average linkage).

3.2.2. Sublabel Map

In addition to the partition, we generate a *sublabel map s*. The sublabel map assigns to each label all labels that are considered to be sublabels of this label. Again, this map can be provided by the user or maintained by the DBMS.

In the example of Figure 3.1, a good sublabel map would be

$$\begin{aligned}
&s(\text{Movie}) := \emptyset \\
&s(\text{Person}) := \{\text{Actor}, \text{Director}\} \\
&s(\text{Actor}) := \emptyset \\
&s(\text{Director}) := \emptyset
\end{aligned}$$

We define the *strict sublabel map* as

$$\begin{aligned}
\widetilde{s} &: \mathcal{D}_L \to \mathcal{P}(\mathcal{D}_L) \\
\widetilde{s}(l) &:= \{l' \in \mathcal{D}_L \mid l' \subseteq l\}
\end{aligned} \tag{3.5}$$

Following this definition, if $l' \in \widetilde{s}(l)$, then it follows logically for all query results Ω that $\mathrm{P}[\Omega](v{:}l \cap v{:}l') = \mathrm{P}[\Omega](v{:}l')$.

Remark 6. Again, the condition for the assumption of a sublabel relationship can be weakened if necessary. A straightforward option is the following definition of the sublabel map

$$s(l) := \{l' \in \mathcal{D}_L \mid \frac{|\sigma_{v:l'}(\sigma_{v:l}(\bigcirc_v))|}{|\sigma_{v:l'}(\bigcirc_v)|} \geq \epsilon\}$$

which considers those labels l' sublabels of l whose nodes are contained in the nodes of l to a (tunable) degree ϵ.

3.3. Solving the Problem of Global Assumptions

We will now show how to address the problem of global assumptions using a label partition D and a sublabel map s.

Suppose a set $L \subseteq \mathcal{D}_L$ of node labels.

If there are two node labels $l_1, l_2 \in L$ with $D(l_1) \neq D(l_2)$, then the label partition tells us we should assume $\mathrm{P}[\bigcirc_v](v{:}l_1 \cap v{:}l_2) = 0$. This implies $\mathrm{P}[\Omega](v{:}l_1 \cap v{:}l_2) = 0$ and therefore

$$\mathrm{P}[\Omega](v{:}L) \approx 0 \ .$$

On the contrary, if all labels from L are in the same member set of D, we make use of the sublabel map to simplify the probability expression. Take a label $l \in L$. If $s(l) \cap L \neq \emptyset$ then the sublabel map tells us to assume that L contains a sublabel $l' \subseteq l$. This means that $\mathrm{P}[\bigcirc_v](v{:}l' \cap v{:}l) \approx \mathrm{P}[\bigcirc_v](v{:}l')$ and implies $\mathrm{P}[\Omega](v{:}l' \cap v{:}l) \approx \mathrm{P}[\Omega](v{:}l')$.

Iterative application of this simplification gives us a smaller set of labels $L' := \{l \in L \mid s(l) \cap L = \emptyset\}$, with

$$\mathrm{P}[\Omega](v{:}L) \approx \mathrm{P}[\Omega](v{:}L') \ .$$

Now we are left with a set of labels L', which contains overlapping labels that are not sublabels of each other. We do now finally assume these labels are independent in Ω, which gives us the solution

$$\mathrm{P}[\Omega](v{:}L) \approx \prod_{l \in L \wedge s(l) \cap L = \emptyset} \mathrm{P}[\Omega](v{:}l)$$

All in all, we can compute the probability of drawing a subgraph where the nodes matched by v have the labels L as

$$\mathrm{P}[\Omega](v{:}L) \approx \begin{cases} 0, \text{ if there are } l_1, l_2 \in L \text{ such that } D(l_1) \neq D(l_2) \\ \prod_{l \in L \wedge s(l) \cap L = \emptyset} \mathrm{P}[\Omega](v{:}l), \text{ otherwise} \end{cases} \tag{3.6}$$

For instance, in the example of Figure 3.1, using the proposed label partition and sublabel map, we have

$$\mathrm{P}[\Omega](v{:}\text{Person} \cap v{:}\text{Actor} \cap v{:}\text{Director}) \approx \mathrm{P}[\Omega](v{:}\text{Actor}) \cdot \mathrm{P}[\Omega](v{:}\text{Director})$$

$$\mathrm{P}[\Omega](v{:}\text{Director} \cap v{:}\text{Movie}) \approx 0$$

4. Logical Properties

In Chapter 2 we have given a formal definition of a graph database and a suitable query algebra. In this Chapter, we explain the Cascades framework for the estimation of so-called logical properties of query results. The result cardinality appears in this framework as one particular logical property.

4.1. Logical vs. Physical Properties

According to Graefe [4], data can have logical and physical properties. Logical properties are properties that can be logically deduced from the data, whereas physical properties depend on the physical environment in which the data is processed.

Logical Database Properties A logical database property is simply a function of the database graph G. We are often interested in functions with some numerical output, e.g. the number of nodes, average density etc. Every DBMS maintains a set of logical database properties which can be used by the query optimizer to estimate result cardinalities.

We write $\mathtt{p}(G)$ to refer to the logical database properties, that are maintained by a particular DBMS with database graph G.

Logical Result Properties A logical result property is simply a function of a query result. The most important logical result property in query optimization is the result cardinality.

We write $\mathtt{p}(\Omega)$ to refer to the set of logical result properties used in a particular DBMS for a query result Ω.

4.2. Estimating Logical Result Properties of Operator Trees

In our graph database model, queries can be expressed as a tree of algebra operators. During query optimization, we want to estimate the result cardinality of such a tree. This estimation has to be done before the query is executed and should be very fast.

In Cascades, the problem of cardinality estimation is decomposed into subproblems, corresponding to the operators in the tree.

The structure of such a subproblem is shown in Figure 4.1. Each operator has a finite number of inputs, which determine the result. Suppose that we know a set of logical properties of each input of an operator. Then we can use this information to estimate the logical properties of the result, including the result cardinality.

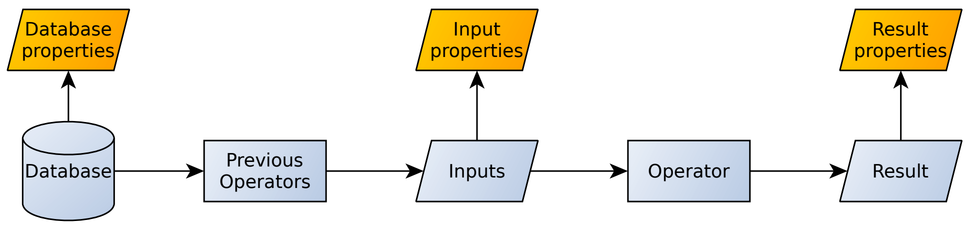


Figure 4.1.: Dependencies between operator inputs, result and logical properties.

The estimated result properties can then be used as the input properties in the next estimation step. The resulting chain of estimations is shown in Figure 4.2.

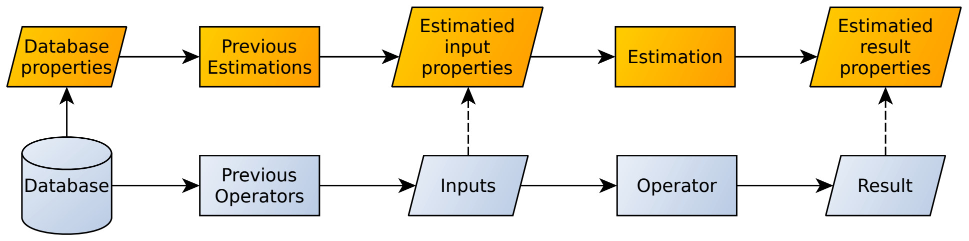


Figure 4.2.: Estimation of result properties.

The more information we put into the logical properties, the more precise the estimations will be. If the information is insufficient, the estimations can diverge from the actual logical properties in a longer estimation chain (symbolized by the dashed arrows in the above Figure). However, storing more information also means increasing memory consumption and processing time. This tradeoff has to be optimized.

In the following definitions, we formalize the idea that was just presented.

Definition 4.2.1 (Estimation function)**.** $E(o)$ is a logical properties estimation function of the operator $o \in \mathcal{O}$ iff it has the following signature:

Input:

1. $\mathtt{p}(G)$, the logical database properties.
2. $\hat{\mathtt{p}}(i_1), \ldots, \hat{\mathtt{p}}(i_n)$, the estimated logical properties of all inputs $i_1, \ldots, i_n$ of o.

Output:

$\hat{\mathtt{p}}(o(i_1, \ldots, i_n))$, an estimation of the logical properties of the output.

Definition 4.2.2 (Estimated logical properties of an operator tree.)**.** We define the estimated logical properties $\hat{\mathtt{p}}(\Omega)$ of an operator tree $\Omega = o(i_1, \ldots, i_n)$ as

$$\hat{\mathtt{p}}(\Omega) := E(o)(\mathtt{p}(G), \hat{\mathtt{p}}(i_1), \ldots, \hat{\mathtt{p}}(i_n))$$

Now the logical properties of an operator tree can be estimated recursively as shown in Figure 4.3.

$$\begin{tikzcd}
\hat{\mathtt{p}}(\sigma_{h:\{\text{Hobby}\}}(\varepsilon_{x \xrightarrow{l:\{\text{LIKES}\}} h}(\sigma_{x:\{\text{Person}\}}(\bigcirc_x)))) \arrow[d, equal] & \\
E(\sigma_{h:\{\text{Hobby}\}}) & \\
E(\varepsilon_{x \xrightarrow{l:\{\text{LIKES}\}} h}) \arrow[u] & \\
E(\sigma_{x:\{\text{Person}\}}) \arrow[u] & \\
E(\bigcirc_x) \arrow[u] & \mathtt{p}(G) \arrow[l] \arrow[ul] \arrow[uul] \arrow[uuul]
\end{tikzcd}$$

Figure 4.3.: Estimation graph for the logical properties of an operator tree.

We will use these Definitions in our Neo4j implementation (see Chapter 5).

5. Cardinality Estimation Model

In this chapter, we use the query algebra defined in Chapter 2 and the logical properties framework defined in Chapter 4 to implement cardinality estimations for Neo4j. Our estimations use a label partition and a sublabel map (cf. Section 3.2) as additional information to the Neo4j statistics.

5.1. The Neo4j Graph Database

Neo4j[1] is a popular graph database written in Java. In Neo4j, a database consists of a *property graph*. A property graph is an extension of a labeled and typed multigraph (cf. Definition 2.1.1) that allows to store properties at nodes. Key to Neo4j's popularity is the expressive and user-friendly query language *Cypher*.

Our query algebra can be used to express an important subset of Cypher, namely the matching of subgraphs with label, type and uniqueness constraints.

Because Neo4j is open-source software and thus easy to modify, we use it as testbed for our cardinality estimations.

5.2. Query Language Restrictions

Due to three limitations, we can only provide estimations for a restricted query space.

Firstly, the estimations developed in this thesis are tested in the JCascades framework. JCascades is a general Java implementation of the Cascades framework for query optimization and is developed at the University of Konstanz. In its current state of development it lacks support for certain kinds of operators, including the NodeJoin operator.

Secondly, to estimate the result cardinalities of cyclic queries (i.e., subgraph patterns containing a cycle), we need statistics about the likeliness of cycles in the database. Unfortunately, Neo4j currently lacks such statistics. Implementing these statistics is not possible as part of this work, due to limited resources and time. Therefore we do not consider cyclic queries, i.e. queries containing a Traverse operator.

Thirdly, in this first version of the estimations, we do not cover the DistinctSelection operator. This is for time reasons only.

As a result, our estimations only cover the incomplete set of operators

$$\{\bigcirc_v, \sigma_{v:l}, \varepsilon_{v \overset{r:T}{\alpha} w}\}$$

[1] `https://neo4j.com/`

The fraction of the result space covered by these operators consists of all query results produced by non-cyclic subgraph patterns having only one single component and no uniqueness constraints. That is, all query results produced by *tree subgraph patterns* with no uniqueness constraints. In Section 5.7 we define the corresponding subset of Cypher.

5.3. Logical Database Properties

The estimations developed in this thesis are intended to be usable in the Neo4j graph database. Therefore, the logical database properties used in the estimations are very similar to those available in the Neo4j statistics.

For convenience we use $*$ as a wildcard label that selects all nodes and similarly as a wildcard relationship type that allows all relationship types. Let $l, l_1, l_2 \in \mathcal{D}_L \cup \{*\}, t \in \mathcal{D}_R \cup \{*\}$ and $\alpha \in \{\leftarrow, \rightarrow\}$. The wildcard label selection is defined as $\sigma_{v:*}(\Omega) := \Omega$.

We require the following logical database properties $\mathtt{p}(G)$.

1. The number of nodes in G having the label l.
$$N(l) := |\sigma_{v:l}(\bigcirc_v)|$$

2. The number of outgoing/incoming relationships of type t in G from nodes with label l_1 to nodes with label l_2 (sometimes called triple statistics).
$$R_\alpha(l_1, t, l_2) := \left|\sigma_{w:l_2}(\varepsilon_{\substack{r:T \\ v\ \alpha\ w}}(\sigma_{v:l_1}(\bigcirc_v)))\right|$$
where $T := \begin{cases} \mathcal{D}_R \text{ if } t = *, \\ \{t\}, \text{ otherwise} \end{cases}$.

3. A label partition D as described in Section 3.2.1.

4. A sublabel map s as described in Section 3.2.2.

Neo4j offers all of these logical database properties, but it estimates
$$R_\alpha(l_1, t, l_2) \approx \min\{R_\alpha(l_1, t, *), R_\alpha(*, t, l_2)\}$$
Our estimations require the actual values of $R_\alpha(l_1, t, l_2)$ to yield good results. Hence, we have to implement the collection of these values. Currently this is done by executing the corresponding queries in Neo4j and storing the result cardinalities in a CSV file. In the future, this should be integrated into the Neo4j statistics and updated online.

Because every relationship has exactly one type, we can obtain the number of relationships of any type in $T \subseteq \mathcal{D}_R$ and any direction α between nodes with label l_1 to nodes with label l_2 as
$$R_\alpha(l_1, T, l_2) = \sum_{t \in T} R_\alpha(l_1, t, l_2)$$

Furthermore, we can define the average degree in α direction as
$$\overline{\deg}_\alpha(l_1, T, l_2) := \frac{R_\alpha(l_1, T, l_2)}{N(l_1)}$$

5.4. Logical Result Properties

As described in Chapter 4 the logical properties of a query result Ω are estimated by building a chain of estimation functions. If the information content of the logical result properties is too small, it can happen that the estimated properties diverge from reality for queries incorporating many operators.

To limit this risk, we store a subgraph pattern $\rho(\Omega)$ producing the query result in the logical properties (in Section 2.4 we have shown that such a subgraph pattern exists for all query results). From this pattern, the estimation functions know about all the previous operators in the estimation chain.

In addition, we store for all node variables $v \in \operatorname{nvar}(\Omega)$ the probability of drawing a subgraph from the query result where the node matched by v has the label l, $\mathrm{P}[\Omega](v{:}l)$ (cf. Equation 3.1).

We also store a Boolean flag, indicating whether the result was produced by the GETNODES operator:

$$\operatorname{initial}(\Omega) :\Leftrightarrow \Omega = \bigcirc_v \text{ for some } v \in \mathcal{X} \tag{5.1}$$

Finally, the logical result properties contain the result cardinality $|\Omega|$. The memory requirements of the logical properties are shown in Table 5.1.

Component	Space complexity
Subgraph pattern	$O(\binom{\lvert R_\rho\rvert}{2}) + \lvert V_\rho\rvert \cdot \lvert\mathcal{D}_L\rvert)$ (cf. Table 2.1)
Label probabilities	$O(\lvert V_\rho\rvert \cdot \lvert\mathcal{D}_L\rvert)$
Initial flag	$O(1)$
Result cardinality	$O(1)$
Total:	$O(\binom{\lvert R_\rho\rvert}{2})) + \lvert V_\rho\rvert \cdot \lvert\mathcal{D}_L\rvert)$

Table 5.1.: Space complexity of the logical result properties.

Remark 7. From a purely logical perspective, the label probabilities, the initial flag and the result cardinality are already determined by the subgraph pattern.

However, the logical result properties are not intended to be logically minimal, but to allow for *efficient* computations in the estimation functions.

Deducing the exact cardinality from the subgraph pattern would often require to execute the query, which we want to avoid. Instead, we want a *fast estimation* of the output cardinality, which is only possible if we store logically redundant information.

5.5. Assumption About Node Degrees

We assume node degrees are approximately uniformly distributed among nodes with the same label, i.e. the individual node degrees are close enough to the average degree to use it as an estimation: For any query result Ω, node variables x, y and relationship directions $\alpha \in \{\leftarrow, \rightarrow\}$ and for any node labels $l_1 \in \mathcal{D}_L, l_2 \in \mathcal{D}_L \cup \{*\}$ we have:

$$\left|\sigma_{y:l_2}(\varepsilon_{x \overset{r:T}{\alpha} y}(\sigma_{x:l_1}(\Omega)))\right| \approx |\sigma_{x:l_1}(\Omega)| \cdot \overline{\deg}_\alpha(l_1, T, l_2) \tag{5.2}$$

For nodes not having any label, we assume the node degree equals the overall average node degree:

$$\left|\sigma_{y:l_2}(\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega \setminus \textstyle\bigcup_{l_1 \in \mathcal{D}_L} \sigma_{x:l_1}(\Omega)))\right| \approx \left|\Omega \setminus \textstyle\bigcup_{l_1 \in \mathcal{D}_L} \sigma_{x:l_1}(\Omega)\right| \cdot \overline{\deg}_\alpha(*, T, l_2) \tag{5.3}$$

5.6. Estimation Functions

An estimation function for the logical result properties of an operator o takes a subgraph pattern, an initial flag, label probabilities and a cardinality corresponding to the input and computes a new subgraph pattern, a new initial flag, new label probabilities and a new result cardinality corresponding to the result of o (see Section 5.4).

We have to define such an estimation function for every operator of our query algebra.

From the proofs that the individual operators are well-defined, we already know how to compute the result subgraph pattern from the input subgraph pattern for every operator (cf. Section 2.3). Therefore, we only have to define functions for the initial flag, the result cardinality and the label probabilities.

5.6.1. Estimation for GetNodes

$\bigcirc_v$ returns the set of all single node subgraphs of G, matched by the variable v.

Initial flag The result is initial:

$$\text{initial}(\bigcirc_v) \Leftrightarrow \texttt{true} \tag{5.4}$$

Result cardinality The result cardinality is the number of nodes in the database:

$$|\bigcirc_v| = N(*) \tag{5.5}$$

Label probabilities The probability of drawing a node matched by v, having a particular label l equals the fraction of nodes in the database having this label:

$$\begin{aligned} \mathrm{P}[\bigcirc_v](v{:}l) &= \frac{|\sigma_{v:l}(\bigcirc_v)|}{|\bigcirc_v|} \\ &= \frac{N(l)}{N(*)} \end{aligned} \tag{5.6}$$

5.6.2. Estimation for NodeLabelSelection

$\sigma_{v:l}(\Omega)$ keeps all subgraphs from Ω, where the node matched by v has the label l.

Initial flag The result of the selection is initial, iff the input is initial and all nodes matched by v in the input have the label l:

$$\begin{aligned} \text{initial}(\sigma_{v:l}(\Omega)) &\Leftrightarrow \sigma_{v:l}(\Omega) = \bigcirc_v \\ &\Leftrightarrow \Omega = \bigcirc_v \wedge \frac{|\sigma_{v:l}(\bigcirc_v)|}{|\bigcirc_v|} = 1 \\ &\Leftrightarrow \text{initial}(\Omega) \wedge \text{P}[\Omega](x{:}l) = 1 \end{aligned} \tag{5.7}$$

Result cardinality The result cardinality of the selection is given as:

$$\begin{aligned} |\sigma_{v:l}(\Omega)| &= \frac{|\sigma_{v:l}(\Omega)|}{|\Omega|} \cdot |\Omega| \\ &= \text{P}[\Omega](v{:}l) \cdot |\Omega| \end{aligned} \tag{5.8}$$

Label probabilities We derive the new label probabilities using a case distinction.

We first look at the label probabilities at the variable v. Firstly, in the result all nodes matched by v have the label l:

$$\begin{aligned} \text{P}[\sigma_{v:l}(\Omega)](v{:}l) &= \frac{|\sigma_{v:l}(\sigma_{v:l}(\Omega))|}{|\sigma_{v:l}(\Omega)|} \\ &= \frac{|\sigma_{v:l}(\Omega)|}{|\sigma_{v:l}(\Omega)|} \\ &= 1 \end{aligned} \tag{5.9}$$

The same holds for all nodes matched by v having a superlabel of l, i.e., $\text{P}[\sigma_{v:l}(\Omega)](v{:}l') = 1$ if $l \in s(l')$.

Now take a label $l' \in \mathcal{D}_L, l' \neq l$. If $D(l') \neq D(l)$, then we assume the labels are disjoint and we have

$$\begin{aligned} \text{P}[\sigma_{v:l}(\Omega)](v{:}l') &= \frac{|\sigma_{v:l'}(\sigma_{v:l}(\Omega))|}{|\sigma_{v:l}(\Omega)|} \\ &= \frac{\text{P}[\Omega](v{:}l' \cap v{:}l)}{\text{P}[\Omega](v{:}l)} \\ &\approx 0 \end{aligned} \tag{5.10}$$

Else, if $l' \in s(l)$, then we assume that l' is a sublabel of l:

$$\begin{aligned} \text{P}[\sigma_{v:l}(\Omega)](v{:}l') &= \frac{\text{P}[\Omega](v{:}l' \cap v{:}l)}{\text{P}[\Omega](v{:}l)} \\ &\approx \frac{\text{P}[\Omega](v{:}l')}{\text{P}[\Omega](v{:}l)} \end{aligned} \tag{5.11}$$

Else, we assume that l and l' are independent in Ω:

$$\begin{aligned}\mathrm{P}[\sigma_{v:l}(\Omega)](v{:}l') &= \frac{\mathrm{P}[\Omega](v{:}l' \cap v{:}l)}{\mathrm{P}[\Omega](v{:}l)} \\ &\approx \frac{\mathrm{P}[\Omega](v{:}l') \cdot \mathrm{P}[\Omega](v{:}l)}{\mathrm{P}[\Omega](v{:}l)} \\ &= \mathrm{P}[\Omega](v{:}l')\end{aligned} \tag{5.12}$$

Now take a node variable v' with $v' \neq v$. The problem is to compute the expression

$$\mathrm{P}[\sigma_{v:l}(\Omega)](v'{:}l') = \frac{\mathrm{P}[\Omega](v'{:}l' \cap v{:}l)}{\mathrm{P}[\Omega](v{:}l)} \tag{5.13}$$

This is impossible without exploiting information about how v and v' are connected in the subgraph pattern ρ. Although we have this information available from ρ in theory, using it would require potentially costly traversals of the pattern and significantly increase the amount of time required to compute the new label probabilities.

Instead, we assume that the events $v'{:}l'$ and $v{:}l$ are independent in Ω, i.e., that selecting a label on one node variable of a pattern does not affect the distribution of labels at other node variables of this pattern:

$$\mathrm{P}[\sigma_{v:l}(\Omega)](v'{:}l') \approx \mathrm{P}[\Omega](v'{:}l') \tag{5.14}$$

Remark 8. This assumption somehow contradicts the motivation of the thesis, because it neglects the possibility of strong correlations between the existence of relationships between nodes and the labels of these nodes.

In future revisions of this work another solution might be proposed. However, the current assumption allows for very local updates to the label probabilities (only the probabilities at the variable v must be updated) and can therefore be computed very fast.

Also note that, correlations between relationship types and labels are considered in the estimation functions of the Expand operator.

5.6.3. Estimation for Expand

$\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega)$ expands the subgraphs in Ω by matching relationships r of types T starting at nodes matched by x and going to nodes matched by y in direction α.

Initial flag The result of an expand is never initial:

$$\begin{aligned}\mathrm{initial}(\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega)) &\Leftrightarrow \varepsilon_{x \overset{r:T}{\alpha} y}(\Omega) = \bigcirc_x \\ &\Rightarrow \mathrm{rvar}(\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega)) = \mathrm{rvar}(\bigcirc_x) \\ &\Rightarrow \{r\} \subseteq \emptyset \\ &\Leftrightarrow \texttt{false}\end{aligned}$$

Result cardinality If the input Ω is initial, i.e. it is the result of a GetNodes operator, we know the exact result cardinality from the database properties:

$$\left|\varepsilon_{x \overset{r:T}{\alpha} y}(\bigcirc_x)\right| = R_\alpha(*, T, *) \tag{5.15}$$

Let us now focus on the case where Ω is not initial. The idea of the estimation function for the result cardinality is to describe the nodes matched by the variable v in Ω in terms of their node labels.

Because there is only a finite number of labels in the database, we can enumerate them according to the partition of overlapping labels: Let D_i be the i-th member set of D and $l_{i,j}$ the j-th label in D_i.

For each node, *one* node label l is chosen that is considered to be most representative for this node. The degree of the node is then estimated as the average degree $\overline{\deg}_\alpha(l, T, *)$ of all nodes having this representative label. The assignment of nodes to representative labels happens using the information stored in the input label probabilities, the label partition D and the sublabel map s.

In the first step, we divide the set of input subgraphs into two subsets. In the first subset, we put those subgraphs where the nodes matched by x have any label. In the second subset, we put the subgraphs where the nodes matched by x have no label:

$$\begin{aligned}\left|\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega)\right| =& \left|\varepsilon_{x \overset{r:T}{\alpha} y}(\textstyle\bigcup_{l \in \mathcal{D}_L} \sigma_{x:l}(\Omega))\right| \quad (1)\\ &+ \\ &\left|\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega \setminus \textstyle\bigcup_{l \in \mathcal{D}_L} \sigma_{x:l}(\Omega))\right| \quad (2)\end{aligned} \tag{5.16}$$

We now rewrite the first term as a sum over the member sets of the node label partition D. After that, we add an inner sum over the respective node labels of each member set. For each node label $l_{i,j} \in D_i$, we count the number of subgraphs produced by the expand applied on the nodes in Ω which have the label $l_{i,j}$, but do not have any of the labels $l_{i,1}, \ldots, l_{i,j-1}$. These nodes are then considered to be best represented by the label $l_{i,j}$. Therefore, their degree is estimated as $\overline{\deg}_\alpha(l_{i,j}, T, *)$, using the degree uniformity assumption. This gives us the following result:

$$
\begin{aligned}
(1) &= \left|\varepsilon_{x\overset{r:T}{\alpha}y}\left(\textstyle\bigcup_{l\in\mathcal{D}_L}\sigma_{x:l}(\Omega)\right)\right| \\
&\approx \sum_{i=1}^{|D|}\left|\varepsilon_{x\overset{r:T}{\alpha}y}\left(\textstyle\bigcup_{j=1}^{|D_i|}\sigma_{x:l_{i,j}}(\Omega)\right)\right| \\
&= \sum_{i=1}^{|D|}\sum_{j=1}^{|D_i|}\left|\varepsilon_{x\overset{r:T}{\alpha}y}\left(\sigma_{x:l_{i,j}}\left(\Omega\setminus\textstyle\bigcup_{k=1}^{j-1}\sigma_{x:l_{i,k}}(\Omega)\right)\right)\right| \\
&\text{Equation 5.2}\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
&\approx \sum_{i=1}^{|D|}\sum_{j=1}^{|D_i|}\left|\sigma_{x:l_{i,j}}\left(\Omega\setminus\textstyle\bigcup_{k=1}^{j-1}\sigma_{x:l_{i,k}}(\Omega)\right)\right|\cdot\overline{\deg}_{\alpha}(l_{i,j},T,*) \\
&= \sum_{i=1}^{|D|}\sum_{j=1}^{|D_i|}|\Omega|\cdot\mathrm{P}[\Omega]\left(x{:}l_{i,j}\cap\overline{\textstyle\bigcup_{k=1}^{j-1}x{:}l_{i,k}}\right)\cdot\overline{\deg}_{\alpha}(l_{i,j},T,*) \\
&= |\Omega|\cdot\sum_{i=1}^{|D|}\sum_{j=1}^{|D_i|}\mathrm{P}[\Omega]\left(x{:}l_{i,j}\cap\textstyle\bigcap_{k=1}^{j-1}\overline{x{:}l_{i,k}}\right)\cdot\overline{\deg}_{\alpha}(l_{i,j},T,*)
\end{aligned}
\tag{5.17}
$$

At this point, we want to highlight a strength of our approach: By taking into account the label probabilities of the input, the estimations are precise even if there are strong dependencies between labels and the presence of relationships. If e.g., the input consists only of nodes having exactly one particular label, then only the average degree of nodes of this label is used to estimate the result cardinality. Neo4j does not track label probabilities and is therefore unable to deal with such dependencies.

The remaining problem is to sensibly estimate $\mathrm{P}[\Omega](x{:}l_{i,j}\cap\bigcap_{k=1}^{j-1}\overline{x{:}l_{i,k}})$, which is the fraction of subgraphs in Ω where the nodes matched by x have the label $l_{i,j}$ but none of the labels $l_{i,1},\ldots,l_{i,j-1}$.

We observe that, if a node has several labels, the representative label is the one that comes first in the ordering in the member set of the label partition. Consequently, this ordering plays an essential role in the quality of the estimation function.

A simple approach is to sort the labels l descendingly by the product $\mathrm{P}(l)\cdot\frac{1}{N(l)}$. This puts the labels in front that cover most of the nodes matched by x in Ω and whose number of nodes in the database is closest to Ω. If this value is similar for two different labels, the label l with higher $\mathrm{P}(l)$ comes first in the ordering. We will use this ordering for the estimation function.

Remark 9. This simple approach does not take into account that nodes can be matched several times by the same variable x in different subgraphs of $\sigma_{x:l}(\Omega)$. It can happen that x matches only one single node in the database in all subgraphs of Ω while $\mathrm{P}(l) = 1$ and $N(l) = |\Omega|$. In this case, the risk of an estimation error is very high, because it is probable that this single node having label l behaves differently than a larger sample of nodes having l.

This problem could be addressed by sorting the labels according to a similarity measure that describes the cardinality of the intersection of the nodes matched by x in $\sigma_{x:l}(\Omega)$ and in $\sigma_{x:l}(\bigcirc_x)$. However, such a similarity measure would require storing and updating information about the uniqueness of nodes matched by a particular node variable and having a particular label.

Now that we have discussed the ordering, we will present a solution for computing $\mathrm{P}[\Omega](x{:}l_{i,j} \cap \bigcap_{k=1}^{j-1} \overline{x{:}l_{i,k}})$.

To make the equations more readable, we write P for $\mathrm{P}[\Omega]$.

Take a partition member set $D_i \in D$ and a subset of labels $L \subset D_i$. Take one label $l \in D_i \setminus L$.

It holds that

$$\begin{aligned}
\mathrm{P}(l \cap \textstyle\bigcap_{l' \in L} \overline{l'}) &= \mathrm{P}(l \mid \textstyle\bigcap_{l' \in L} \overline{l'}) \cdot \mathrm{P}(\textstyle\bigcap_{l' \in L} \overline{l'}) \\
&= (1 - \mathrm{P}(\bar{l} \mid \textstyle\bigcap_{l' \in L} \overline{l'})) \cdot \mathrm{P}(\textstyle\bigcap_{l' \in L} \overline{l'}) \\
&= (1 - \frac{\mathrm{P}(\bigcap_{l' \in L \cup \{l\}} \overline{l'})}{\mathrm{P}(\bigcap_{l' \in L} \overline{l'})}) \cdot \mathrm{P}(\textstyle\bigcap_{l' \in L} \overline{l'}) \\
&= \mathrm{P}(\textstyle\bigcap_{l' \in L} \overline{l'}) - \mathrm{P}(\textstyle\bigcap_{l' \in L \cup \{l\}} \overline{l'})
\end{aligned} \tag{5.18}$$

Take again a subset of labels $L \subset D_i$ and a label $l \in D_i \setminus L$. For all $l' \in L$ that are sublabels of l, the probability of drawing a node that does neither have the label l nor the label l' equals the probability of drawing a node that does not have the label l:

$$\mathrm{P}(\bar{l} \cap \overline{l'}) \approx \mathrm{P}(\bar{l})$$

Consequently, let $S := (L \setminus s(l)) \cup \{l\}$. We have

$$\mathrm{P}(\textstyle\bigcap_{l' \in L \cup \{l\}} \overline{l'}) \approx \mathrm{P}(\textstyle\bigcap_{l' \in S \cup \{l\}} \overline{l'}) \tag{5.19}$$

Iterative application of Equation 5.19 gives us a minimal set of superlabels S^*. For all $l \in S^*$, no sublabel of l is contained in S^*, i.e., $S^* \cap s(l) = \emptyset$. For this set of superlabels we assume independence and obtain

$$\mathrm{P}(\textstyle\bigcap_{l' \in S^*} \overline{l'}) \approx \prod_{l' \in S^*} 1 - \mathrm{P}(l') \tag{5.20}$$

Together, the Equations 5.18 and 5.20 provide an easily computable, recursive estimation of $\mathrm{P}[\Omega](x{:}l \cap \bigcap_{l' \in L} \overline{x{:}l'})$.

For the second term we apply the same technique:

$$
\begin{aligned}
(2) &= \left|\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega \setminus \textstyle\bigcup_{l \in \mathcal{D}_L} \sigma_{x:l}(\Omega))\right| \\
&\text{Equation 5.3} \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
&\approx \left|\Omega \setminus \textstyle\bigcup_{l \in \mathcal{D}_L} \sigma_{x:l}(\Omega)\right| \cdot \overline{\deg}_\alpha(*, T, *) \qquad (5.21) \\
&= |\Omega| \cdot \mathrm{P}[\Omega](\overline{\textstyle\bigcup_{l \in \mathcal{D}_L} x{:}l}) \cdot \overline{\deg}_\alpha(*, T, *) \\
&= |\Omega| \cdot (1 - \mathrm{P}[\Omega](\textstyle\bigcap_{l \in \mathcal{D}_L} x{:}l)) \cdot \overline{\deg}_\alpha(*, T, *)
\end{aligned}
$$

From Equation 5.20 we know how to recursively compute the probability expression in this formula.

To demonstrate the computability of the expand estimations, we use Algorithm 1 that computes both terms (1) and (2) in a single loop over the set of node labels.

The most interesting variables are the following:

- *estDeg* stores the amount contributed to the total estimated degree by the nodes having one of the already covered labels (*coveredLabels*).
- *remaining* stores the fraction of nodes matched by v whose degree has not yet been estimated. After the loop has finished, this variable stores the fraction of nodes matched by v not having any label, which allows to compute (2).
- *superLabels* stores the current minimal set of superlabels, as defined above. It is updated once in each iteration.
- *notCoveredBySuperLabels* stores the fraction of nodes matched by v that does not have any of these superlabels. If the minimal set of superlabels changes because of the current label, this variable has to be recomputed as a product over all superlabels. Otherwise, the cached value of the last iteration can be used and only a single multiplication is needed.

To better understand this algorithm, recall the example database about movies and its distribution of node labels shown in Figure 3.1.

In Figure 5.1 we add to this diagram the set of nodes matched by the variable v of some query result Ω. Note that the size of the area does not necessarily correspond to the cardinality of Ω, as nodes can be matched several times by the same variable in different subgraphs.

The outer loop of the algorithm iterates over the member sets of the label partition. The inner loop iterates over the labels of one member set. The order is given by the indices of the label variable $l_{i,j}$. Suppose the algorithm iterates over the label partition in the following order:

$$
\begin{aligned}
l_{1,1} &= \text{Movie} \\
l_{2,1} &= \text{Actor} \\
l_{2,2} &= \text{Director} \\
l_{2,3} &= \text{Person}
\end{aligned}
$$

Algorithm 1: COMPUTEEXPANDCARDINALITY

Input: $\hat{\mathrm{p}}(\Omega)$ (estimation of the logical result properties of Ω), $\mathrm{p}(G)$ (logical database properties)

Output: estimation of $\left|\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega)\right|$

```
1  if initial($\Omega$) then
       // We know the exact cardinality from the database properties.
2      return $R_\alpha(*, T, *)$
3  else
       // We estimate the cardinality by combining the node degrees for
       the different labels.
4      $estDeg \leftarrow 0$
5      $remaining \leftarrow 1$
       // (1) Estimate the node degree for nodes having a label.
6      $i \leftarrow 1$
7      while $i < |D| \land remaining > 0$ do
           // Outer sum over the partition member sets.
8          $oldRemaining \leftarrow remaining$
9          $superLabels \leftarrow \emptyset$
10         $notCoveredBySuperLabels \leftarrow 1$
11         $coveredLabels \leftarrow \emptyset$
12         $j \leftarrow 1$
13         while $j < |D_i| \land remaining > 0$ do
               // Inner sum over the labels in a partition member set.
14             if $l_{i,j} \notin coveredLabels$ then
15                 $superLabels \leftarrow (superLabels \setminus s(l_{i,j}))$
16                 $changed \leftarrow$ whether $superLabels$ has changed
17                 $superLabels \leftarrow superLabels \cup \{l_{i,j}\}$
18                 if $changed$ then
19                     $notCoveredBySuperLabels \leftarrow \prod_{l \in superLabels} 1 - \mathrm{P}[\Omega](x{:}l)$
20                 else
21                     $notCoveredBySuperLabels \leftarrow$
                       $notCoveredBySuperLabels \cdot (1 - \mathrm{P}[\Omega](x{:}l_{i,j}))$
22                 $newRemaining \leftarrow notCoveredBySuperLabels - (1 - oldRemaining)$
23                 $estDeg \leftarrow estDeg + \overline{\deg}_\alpha(l_{i,j}, T, *) \cdot (remaining - newRemaining)$
24                 $remaining \leftarrow newRemaining$
25             $coveredLabels \leftarrow coveredLabels \cup s(l_{i,j})$
26             $j \leftarrow j + 1$
27         $i \leftarrow i + 1$
       // (2) Estimate the node degree for nodes having no label.
28     $estDeg \leftarrow estDeg + \overline{\deg}_\alpha(*, T, *) \cdot remaining$
       // Compute the result cardinality.
29     return $|\Omega| \cdot estDeg$
```

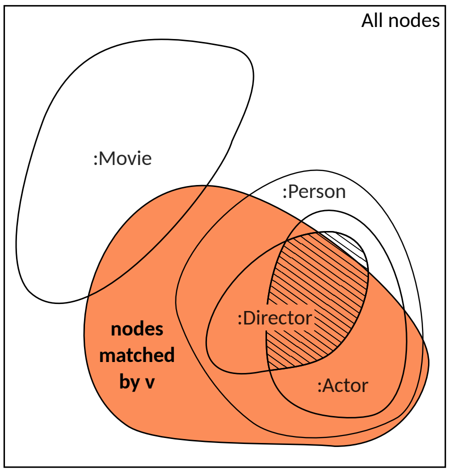
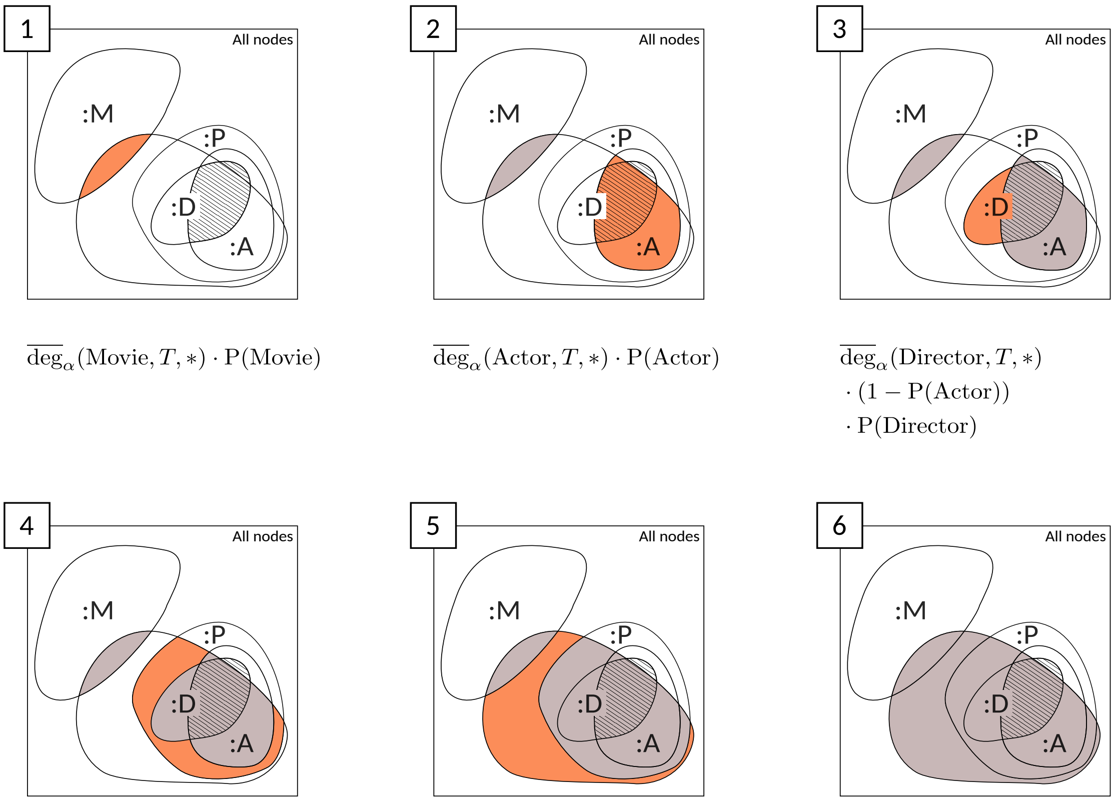


Figure 5.1.: Distribution of node labels and a query result in a example database.

We write again $\mathrm{P}(l)$ instead of $\mathrm{P}[\Omega](v{:}l)$ for better readability. The trace of the algorithm execution of the example is given by Table 5.2 and Table 5.3. Reading the tables from top to bottom we can see all variable assignments performed by the algorithm in chronological order. The column $l_{i,j}$ shows, which label is currently used to estimate the degree of a fraction of the matched nodes.

Figure 5.2 shows the usage of representative labels for the nodes matched by v as a sequence of Venn diagrams. In each diagram, the orange area represents the fraction of nodes whose degree is estimated in the current iteration of the algorithm. The result of this estimation is written below the diagram (cf. the trace of the algorithm in Table 5.2 and Table 5.3). The total estimated degree of the nodes matched by v is the sum of all the partial degrees shown in the individual diagrams.



1: $\overline{\deg}_\alpha(\text{Movie}, T, *) \cdot \text{P}(\text{Movie})$

2: $\overline{\deg}_\alpha(\text{Actor}, T, *) \cdot \text{P}(\text{Actor})$

3: $\overline{\deg}_\alpha(\text{Director}, T, *) \cdot (1 - \text{P}(\text{Actor})) \cdot \text{P}(\text{Director})$

4: $\overline{\deg}_\alpha(\text{Person}, T, *) \cdot [(1 - \text{P}(\text{Actor})) \cdot (1 - \text{P}(\text{Director})) - (1 - \text{P}(\text{Person}))]$

5: $\overline{\deg}_\alpha(*, T, *) \cdot [(1 - \text{P}(\text{Person})) - \text{P}(\text{Movie})]$

Figure 5.2.: Step-by-step visualization of the expand cardinality estimation. *Orange:* The fraction of nodes whose degree is currently estimated. *Grey:* Already covered fraction of nodes. *Below each diagram:* The estimated degree corresponding to the orange fraction.

Table 5.2.: Trace of the ComputeExpandCardinality algorithm (1).

i	*j*	$l_{i,j}$	*estDeg*	*remaining*	*oldRemaining*	*superLabels*	*coveredLabels*	*notCoveredBySuperLabels*	*changed*	*newRemaining*
			0							
				1						
1										
					1					
						$\emptyset$				
								1		
							$\emptyset$			
	1	**Movie**								
						$\emptyset$				
									$false$	
						{Movie}				
								1 − P(Movie)		
										1 − P(Movie)
			$\overline{\deg}_\alpha(\text{Movie}, T, *) \cdot \text{P(Movie)}$							
				1 − P(Movie)						
							{Movie}			
	2									
2										
						$\emptyset$				
								1		
							$\emptyset$			
	1	**Actor**								
								1		
						$\emptyset$				
									$false$	
						{Actor}				
								1 − P(Actor)		
										(1 − P(Actor)) − P(Movie)
			$\overline{\deg}_\alpha(\text{Movie}, T, *) \cdot \text{P(Movie)}$ $+ \overline{\deg}_\alpha(\text{Actor}, T, *) \cdot \text{P(Actor)}$							
				(1 − P(Actor)) − P(Movie)						
							{Actor}			

Table 5.3.: Trace of the ComputeExpandCardinality algorithm (2).

i	*j*	$l_{i,j}$	*estDeg*	*remaining*	*oldRemaining*	*superLabels*	*coveredLabels*	*notCoveredBySuperLabels*	*changed*	*newRemaining*
	2	**Director**								
						{Actor}				
									false	
						{Actor, Director}				
								$(1 - \mathrm{P}(\mathrm{Actor})) \cdot (1 - \mathrm{P}((\mathit{Director})))$		
										$(1 - \mathrm{P}(\mathrm{Actor})) \cdot (1 - \mathrm{P}(\mathrm{Director})) - \mathrm{P}(\mathrm{Movie})$
			$\overline{\deg}_\alpha(\mathrm{Movie}, T, *) \cdot \mathrm{P}(\mathrm{Movie}) + \overline{\deg}_\alpha(\mathrm{Actor}, T, *) \cdot \mathrm{P}(\mathrm{Actor}) + \overline{\deg}_\alpha(\mathrm{Director}, T, *) \cdot (1 - \mathrm{P}(\mathrm{Actor})) \cdot \mathrm{P}(\mathrm{Director})$							
				$(1 - \mathrm{P}(\mathrm{Actor})) \cdot (1 - \mathrm{P}(\mathrm{Director})) - \mathrm{P}(\mathrm{Movie})$						
							{Actor, Director}			
	3	**Person**								
						∅				
									true	
						{Person}				
								$1 - \mathrm{P}(\mathrm{Person})$		
										$(1 - \mathrm{P}(\mathrm{Person})) - \mathrm{P}(\mathrm{Movie})$
			$\overline{\deg}_\alpha(\mathrm{Movie}, T, *) \cdot \mathrm{P}(\mathrm{Movie}) + \overline{\deg}_\alpha(\mathrm{Actor}, T, *) \cdot \mathrm{P}(\mathrm{Actor}) + \overline{\deg}_\alpha(\mathrm{Director}, T, *) \cdot (1 - \mathrm{P}(\mathrm{Actor})) \cdot \mathrm{P}(\mathrm{Director}) + \overline{\deg}_\alpha(\mathrm{Person}, T, *) \cdot [(1 - \mathrm{P}(\mathrm{Actor})) \cdot (1 - \mathrm{P}(\mathrm{Director})) - (1 - \mathrm{P}(\mathrm{Person}))]$							
				$(1 - \mathrm{P}(\mathrm{Person})) - \mathrm{P}(\mathrm{Movie})$						
							{Actor, Director, Person}			
	4									
3										
			$\overline{\deg}_\alpha(\mathrm{Movie}, T, *) \cdot \mathrm{P}(\mathrm{Movie}) + \overline{\deg}_\alpha(\mathrm{Actor}, T, *) \cdot \mathrm{P}(\mathrm{Actor}) + \overline{\deg}_\alpha(\mathrm{Director}, T, *) \cdot (1 - \mathrm{P}(\mathrm{Actor})) \cdot \mathrm{P}(\mathrm{Director}) + \overline{\deg}_\alpha(\mathrm{Person}, T, *) \cdot [(1 - \mathrm{P}(\mathrm{Actor})) \cdot (1 - \mathrm{P}(\mathrm{Director})) - (1 - \mathrm{P}(\mathrm{Person}))] + \overline{\deg}_\alpha(*, T, *) \cdot [(1 - \mathrm{P}(\mathrm{Person})) - \mathrm{P}(\mathrm{Movie})]$							

In the beginning, the counter of the outer loop of the algorithm points to the first partition member set of the label partition. The counter of the inner loop points to the first label in this member set, which is $l_{1,1}$ = Movie. The algorithm now estimates the amount contributed to the total degree by nodes having the "Movie" label. In Diagram 1 in Figure 5.2 the area corresponding to these movie nodes is highlighted in orange. The algorithm correctly estimates the fraction of these nodes as P(Movie). For any movie node, the algorithm assumes a degree of $\overline{\deg}_\alpha(l_{1,1}, T, *) = \overline{\deg}_\alpha(\text{Movie}, T, *)$. Therefore it adds $\overline{\deg}_\alpha(\text{Movie}, T, *) \cdot \text{P(Movie)}$ to the accumulator variable *estDeg*.

Next, the inner loop terminates, because all labels of the first partition member set $\{Movie\}$ have been processed. The algorithm now starts processing the second partition member set.

The first label in the second partition member set is $l_{2,1}$ = Actor. Diagram 2 in Figure 5.2 shows the fraction of nodes matched by v that are actors, but not movies. The algorithm correctly estimates this fraction as P(Actor), ignoring the movie label from the previous partition member set (movies and actors are disjoint, so no actor node can be a movie node). It adds $\overline{\deg}_\alpha(\text{Actor}, T, *) \cdot \text{P(Actor)}$ to *estDeg*.

Next, the algorithm processes the label $l_{2,2}$ = Director. Diagram 3 in Figure 5.2 shows the fraction of nodes matched by v that are directors, but neither actors nor movies. The algorithm estimates this fraction as $f := (1 - \text{P(Actor)}) \cdot \text{P(Director)}$, because it assumes independence between labels that are in the same partition member set and not in a sublabel relation. It adds $\overline{\deg}_\alpha(\text{Director}, T, *) \cdot f$ to *estDeg*.

Next, the algorithm processes the label $l_{2,3}$ = Person. Diagram 4 in Figure 5.2 shows the fraction of nodes matched by v that are persons, but neither directors, nor actors, nor movies. Because "Director" and "Actor" are sublabels of "Person", the algorithm subtracts the fraction of nodes having the labels "Director" or "Actor" from the fraction of nodes having the label "Person". It correctly estimates the fraction as

$$\begin{aligned} f &:= [(1 - \text{P(Actor)}) \cdot (1 - \text{P(Director)}) - (1 - \text{P(Person)})] \\ &= \text{P(Person)} - 1 + 1 - \text{P(Director)} - \text{P(Actor)} + \text{P(Director)} \cdot \text{P(Actor)} \\ &= \text{P(Person)} - \text{P(Actor} \cup \text{Director)} \end{aligned}$$

and adds $\overline{\deg}_\alpha(\text{Person}, T, *) \cdot f$ to *estDeg*.

Now both the inner loop and the outer loop terminate, because there are no more labels to visit. The remaining fraction of nodes does not have any label and is highlighted in Diagram 5 in Figure 5.2. The algorithm already has an estimate of this fraction with its *remaining* variable. It estimates the degree of the nodes in this fraction as $\overline{\deg}_\alpha(*, T, *)$ and therefore adds $\overline{\deg}_\alpha(*, T, *) \cdot remaining$ to *estDeg*.

Finally, the algorithm multiplies the accumulated degree with the input size and returns $|\Omega| \cdot estDeg$.

Label probabilities It holds that for all $z \in \operatorname{nvar}(\Omega)$ and labels l

$$\begin{aligned}
\mathrm{P}[\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega)](z{:}l) &= \frac{\left|\sigma_{z:l}(\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega))\right|}{\left|\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega)\right|} \\
&= \frac{\left|\varepsilon_{x \overset{r:T}{\alpha} y}(\sigma_{z:l}(\Omega))\right|}{\left|\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega)\right|}
\end{aligned} \tag{5.22}$$

We can already compute this expression using the estimation function for the expand cardinality.

For the new variable $y \notin \operatorname{nvar}(\Omega)$, we can proceed analogously to how we estimated the result size. For all $l \in \mathcal{D}_L$:

$$\begin{aligned}
\left|\sigma_{y:l}(\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega))\right| &= \left|\sigma_{y:l}(\varepsilon_{x \overset{r:T}{\alpha} y}(\textstyle\bigcup_{l \in \mathcal{D}_L} \sigma_{x:l}(\Omega)))\right| \\
&\quad + \left|\sigma_{y:l}(\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega \setminus \textstyle\bigcup_{l \in \mathcal{D}_L} \sigma_{x:l}(\Omega)))\right| \\
&\text{Equation 5.2 } \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
&\approx |\Omega| \cdot \Big(\sum_{i=1}^{|D|} \sum_{j=1}^{|D_i|} \mathrm{P}[\Omega](x{:}l_{i,j} \cap \textstyle\bigcap_{k=1}^{j-1} \overline{x{:}l_{i,k}}) \cdot \overline{\deg}_\alpha(l_{i,j}, T, l) \\
&\quad + (1 - \mathrm{P}[\Omega](\textstyle\bigcap_{l \in \mathcal{D}_L} x{:}l)) \cdot \overline{\deg}_\alpha(*, T, l)\Big)
\end{aligned} \tag{5.23}$$

This allows us to compute

$$\mathrm{P}[\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega))](y{:}l) = \frac{\left|\sigma_{y:l}(\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega))\right|}{\left|\varepsilon_{x \overset{r:T}{\alpha} y}(\Omega)\right|} \tag{5.24}$$

5.7. Mapping from Cypher to an Operator Tree

In order to use our estimations in Neo4j, we have to map a Cypher query to a tree of algebra operators.

Definition 5.7.1 (Single pattern query in Cypher)**.** The following context-free grammar produces all *single pattern queries* in the Cypher language that return the whole matched subgraphs (no projection). We write CSP for Cypher single pattern.

The grammar is given in EBNF and has the start symbol CSP. We write $NT \in S$ if the non-terminal symbol NT can be substituted with any of the terminal symbols in a finite set S.

CSP → "MATCH" Path "RETURN *"
Path → Node | Path Rel Path | Path " ," Path
Node → "(" NodeVar { ":" NodeLabel } ")"
Rel → "-[" RelVar { ":" RelType } "]->" | "<-[" RelVar { ":" RelType } "]-"
$NodeVar \in \mathcal{X}$
$NodeLabel \in \mathcal{D}_L$
$RelVar \in \mathcal{X}$
$RelType \in \mathcal{D}_R$

Example 5.7.1. Recall once more the movie database example from Figure 3.1.

```
MATCH (x:Actor)-[r:ACTS_IN]->(m:Movie)<-[s:ACTS_IN]-(y:Actor)
RETURN *
```

is a CSP query on this database, returning all pairs of actors acting in the same movie.

Given a CSP query, generating a corresponding subgraph pattern is straightforward:

1. V_ρ consists of all variables surrounded by round brackets.
2. R_ρ consists of all variables surrounded by square brackets.
3. $\lambda_\rho(r)$ is the pair of node variables that are left and right of the relationship variable r in the query. The order of the node variables in the pair depends on the direction of the ASCII arrow in the query.
4. $\mathtt{l}_\rho(v)$ consists of all node labels surrounded by the same round brackets as v.
5. $\mathtt{t}_\rho(r)$ consists of all relationship types surrounded by the same square brackets as r.
6. $\mathtt{u}_\rho := \{\{r, s\} \mid r, s \in R_\rho \wedge r \neq s\}$

To estimate the result cardinality of a CSP query, we can map it to a subgraph pattern, find a corresponding operator tree and then apply the estimation functions of the operators.

However, for various reasons already explained in Section 5.2, we only have estimation functions for the three operators GetNodes, LabelSelection and Expand. We do not yet have an estimation function for the operators Join, Traverse and DistinctSelection. This causes two major limitations of this thesis.

1. Due to the lack of estimation functions for the Join and Traverse operators, we can only estimate the result cardinality of those CSP queries that map to a tree subgraph pattern.
2. The lack of an estimation function for the DistinctSelection operator means we cannot estimate the effect of uniqueness constraints on relationships.

According to the Cypher specification, all pairs of relationship variables are required to be unique in a CSP query[2] (see step 6 above). Currently, we ignore this requirement, by mapping a CSP query to an operator tree that does not contain a DistinctSelection and is thus not semantically equivalent to the query.

Nevertheless, the result cardinality of this mapped operator tree is an upper bound of the result cardinality of the CSP query. Our statistical evaluation in the next Chapter shows that using this upper bound is already a good estimation (cf. Section 6.2.2).

[2]`https://neo4j.com/docs/developer-manual/3.1/cypher/introduction/uniqueness/`, 20.05.17

6. Cardinality Estimation Model (Experimental Evaluation)

In Chapter 5 we have defined estimations for the cardinalities of query results in a graph database and explained how they can be implemented in Neo4j. In this Chapter, we evaluate our implementation using a benchmark and compare it with Neo4j's current implementation.

6.1. Test setting

We compare our cardinality estimations with the estimations of Neo4j version "3.1.0-M08" (Maven version)[1].

Our test takes a database and a catalog of queries on this database as input. It outputs a CSV-file containing the actual result cardinality of the query and the estimated cardinality of our and Neo4j's estimation model:

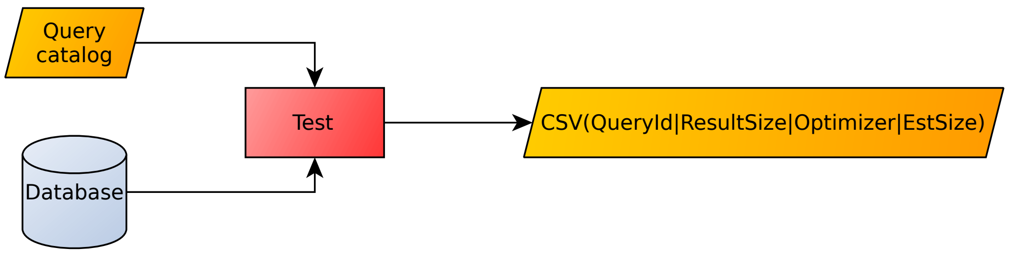


Figure 6.1.: The test setting.

The platform of execution is not relevant, because the measured variables (result size and estimated sizes) are platform-independent.

[1] `https://mvnrepository.com/artifact/org.neo4j/neo4j/3.1.0-M08`

6.1.1. Query Dimensions

As discussed in Section 5.7, our cardinality estimations cover all CSP queries that map to a tree subgraph pattern. Within this query space, we can group queries according to several important dimensions:

1. Query shapes
   - 1.1. Chain queries (denoted by –)
   - 1.2. Star queries (denoted by *)
   - 1.3. Snowflake queries (denoted by #)
2. Type constraints
   - 2.1. Queries without type constraints
   - 2.2. Queries with type constraints
3. Label constraints
   - 3.1. Queries without label constraints
   - 3.2. Queries with label constraints
     - 3.2.1. Queries with redundant label constraints (explanation below)
     - 3.2.2. Queries where labels are in a sublabel relation
     - 3.2.3. Queries where labels are disjoint
     - 3.2.4. Queries where labels are (approximately) independent

We call a label constraint on a node variable *redundant*, if the label is already determined by the type of a relationship connected to the node variable. E.g. in the SNB schema shown below, if a node variable is connected to an outgoing relationship of type "HAS_INTEREST", adding "Person" as a label constraint on this node variable is redundant.

In order to be representative, a catalog of test queries should cover most of the value combinations of the given dimensions.

6.1.2. Dataset: The LDBC Social Network Benchmark

According to the homepage:

> "The Social Network Benchmark (SNB) consists of a data generator that generates a synthetic social network, used in three workloads: Interactive, Business Intelligence and Graph Analytics. Currently, only the Interactive Workload has been released in draft stage."
>
> (`http://ldbcouncil.org/developer/snb`, 06/04/2017)

We use the dataset of the Interactive workload [2] with default settings.

Importing the Dataset

After executing the data generator[2], the generated CSV files have to be imported into Neo4j using the Neo4j import tool[3]. The import requires a reformatting of the CSV files, which can be achieved by Jonathan Ellithorpe's *DataFormatConverter*[4].

Label Partition and Sublabel Map

The schema of the SNB data is shown in Figure 6.2. The mapping to a property graph performed by the `DataFormatConverter` uses a property "type" to differentiate between the subclasses of the Place and Organisation classes. Because we do not consider properties in our graph model, we add the respective labels (City, Country, Continent, University, Company) to the subclasses after the import. After this step, all classes in the schema map directly to node labels in our database.

Observe that every node in the database has at least one label. Furthermore, any two different labels in the SNB schema are either disjoint or in a sublabel relation. This label distribution can be stored in a label partition and a sublabel map, which are needed by our estimations. We use the strict label partition and the strict sublabel map (cf. Section 3.2.2). The strict label partition is given as

$$\begin{aligned}
D_{\texttt{SNB}} := \{&\{\text{Message}, \text{Post}, \text{Comment}\},\\
&\{\text{Place}, \text{Continent}, \text{Country}, \text{City}\},\\
&\{\text{Organisation}, \text{University}, \text{Company}\},\\
&\{\text{Person}\},\\
&\{\text{Forum}\},\\
&\{\text{Tag}\},\\
&\{\text{TagClass}\}\}
\end{aligned} \tag{6.1}$$

[2]`https://github.com/ldbc/ldbc_snb_datagen`, 1.6.2017
[3]`https://neo4j.com/docs/operations-manual/3.1/tools/import/`, 5.6.2017
[4]`https://github.com/PlatformLab/ldbc-snb-impls/blob/6d52ca5cc492ca8aeffed4cbbdcf7d1a6c073061/snb-interactive-neo4j/src/main/java/net/ellitron/ldbcsnbimpls/interactive/neo4j/util/DataFormatConverter.java`, 5.6.2017

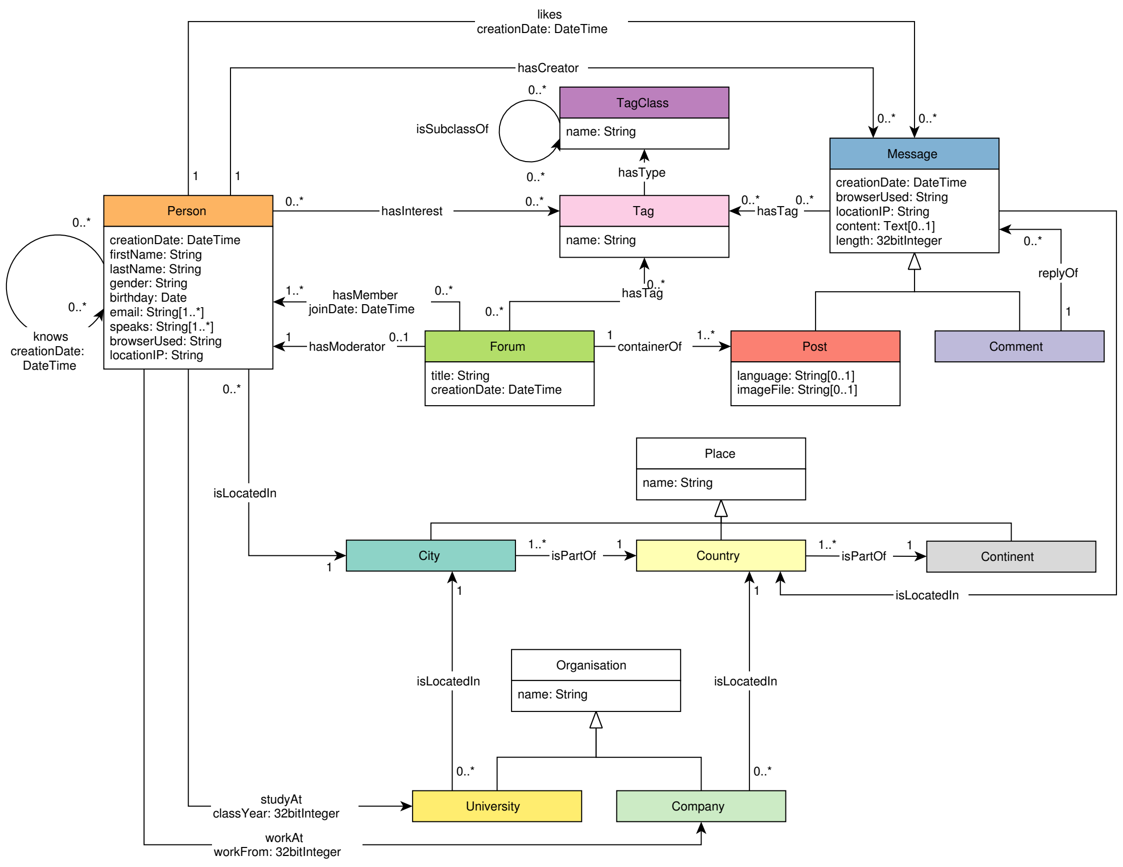


Figure 6.2.: Schema of the SNB Interactive workload. Taken from *LDBC SNB Documentation 0.3.0*, `https://github.com/ldbc/ldbc_snb_docs`, 06/04/2017.

The strict sublabel map is given as

$$\begin{aligned}
&s_{\texttt{SNB}}(\text{Message}) := \{\text{Post}, \text{Comment}\},\\
&s_{\texttt{SNB}}(\text{Post}) := \emptyset\\
&s_{\texttt{SNB}}(\text{Comment}) := \emptyset\\
&s_{\texttt{SNB}}(\text{Place}) := \{\text{Continent}, \text{Country}, \text{City}\},\\
&s_{\texttt{SNB}}(\text{Continent}) := \emptyset\\
&s_{\texttt{SNB}}(\text{Country}) := \emptyset\\
&s_{\texttt{SNB}}(\text{City}) := \emptyset\\
&s_{\texttt{SNB}}(\text{Organisation}) := \{\text{University}, \text{Company}\},\\
&s_{\texttt{SNB}}(\text{University}) := \emptyset\\
&s_{\texttt{SNB}}(\text{Company}) := \emptyset\\
&s_{\texttt{SNB}}(\text{Person}) := \emptyset\\
&s_{\texttt{SNB}}(\text{Forum}) := \emptyset\\
&s_{\texttt{SNB}}(\text{Tag}) := \emptyset\\
&s_{\texttt{SNB}}(\text{TagClass}) := \emptyset
\end{aligned} \tag{6.2}$$

6.1.3. Query Catalog

The queries contained in the SNB Interactive workload exceed by far what we can express in our current query algebra. Therefore, we select a subset of interesting queries in the SNB schema and compare the cardinality estimations of our model and Neo4j's model on this subset.

Tables 6.1, 6.2 and 6.3 show our query catalog for the SNB. For each query, the table lists the unique query id, the Cypher pattern, the query shape and whether the query has type or label constraints.

Our query catalog covers most of the value combinations of the query dimensions specified in Section 6.1.1. However, due to the SNB schema, it does not occur that two labels are (approximately) independent.

QueryId	Pattern	Shape	Type constraints	Label constraints
0	`(a)`	--		
1	`(a)-->(b)`	--		
2	`(a)-->(b)-->(c)`	--		
3	`(a:Person)-->(b:Tag)<--(c:Forum)`	--		yes
4	`(a:Person)-->(b:Tag)<--(c:Forum)-->(d:Person)`	--		yes
5	`(a)-[:LIKES]->(b)-[:HAS_CREATOR]->(c)`	--	yes	
6	`(a)-[:CONTAINER_OF]->(b)-[:HAS_TAG]->(c)`	--	yes	
7	`(b)<-[:CONTAINER_OF]-(c)-[:HAS_MODERATOR]->(a)-[:HAS_INTEREST]->(d)`	--	yes	
8	`(a)-[:HAS_CREATOR]->(b)<-[:HAS_MODERATOR]-(c)`	--	yes	
9	`(x)<--(y)<-[:IS_LOCATED_IN]-(z)`	--	yes	
10	`(a:Person)-[:LIKES]->(b:Message)-[:HAS_CREATOR]->(c:Person)`	--	yes	yes (redundant)
11	`(a:Forum)-[:CONTAINER_OF]->(b:Post)-[:HAS_TAG]->(c:Tag)`	--	yes	yes (redundant)
12	`(b:Post)<-[:CONTAINER_OF]-(c:Forum)-[:HAS_MODERATOR]->(a:Person),` `(a)-[:HAS_INTEREST]->(d:Tag)`	--	yes	yes (redundant)
13	`(a:Comment)-[:HAS_CREATOR]->(b:Person)<-[:HAS_MODERATOR]-(c:Forum)`	--	yes	yes
14	`(p:Person:Post)`	--		yes (disjoint)
15	`(x)-->(p:Comment:University)<--(y)`			yes (disjoint)
16	`(c)-->(d)-->(e:Country:Continent)`			yes (disjoint)
17	`(x:Message:Post)`	--		yes (sublabels)
18	`(x:Company:Organisation)<-[:WORK_AT]-(p)`	--	yes	yes (sublabels)
19	`(p)-[:IS_LOCATED_IN]->(x:Continent:Place)`	--	yes	yes (sublabels)
20	`(p2:Person)-->(p1)-[:LIKES]->(c)-[:REPLY_OF]->(x)`	--	yes	yes
21	`(c3)-[:IS_PART_OF]->(c2)<-[:IS_PART_OF]-(c1:Country)`	--	yes	yes
22	`(c4)-[:IS_LOCATED_IN]->(c3)-[:IS_PART_OF]->(c2)<-[:IS_PART_OF]-(c1:Country)`	--	yes	yes
23	`(p1)-[:HAS_INTEREST]->(t)-[:IS_LOCATED_IN]->(p2)`	--	yes	
24	`(c1)-[:REPLY_OF]->(c2)-[:STUDY_AT]->(u)`	--	yes	
25	`(p:Person)-[:IS_LOCATED_IN]->(x:Continent:Place)`	--	yes	yes
26	`(c)<-[:HAS_TYPE]-(t)-->(x)`	--	yes	

Table 6.1.: The catalog of test queries for the SNB (part 1).

QueryId	Pattern	Shape	Type constraints	Label constraints
27	`(a)-->(b), (a)-->(c)`	*		
28	`(a:Person)-->(b), (a)-->(c:Person)`	*		yes
29	`(a:Message)-->(b), (a)-->(c), (a)-->(d), (a)-->(e:Message)`	*		yes
30	`(a)-->(b), (a)<--(c:Company)`	*		yes
31	`(a)-->(b:Post), (a)-->(c:Tag), (a)<--(d:City)`	*		yes
32	`(a)-->(b:Continent), (a)-->(c:Place), (a)-->(d:Place), (a)-->(e:Place)`	*		yes
33	`(m)-[:IS_LOCATED_IN]->(l), (m)-[:REPLY_OF]->(m2), (m)-[:HAS_CREATOR]->(p)`	*	yes	yes
34	`(f)-[:HAS_MODERATOR]->(mod), (f)-[:HAS_MEMBER]->(mem), (f)-[:HAS_TAG]->(x)`	*	yes	yes
35	`(c1)-[:REPLY_OF]->(c2)-[:STUDY_AT]->(u), (c2)-->(x1), (x2)-->(c2)`	*	yes	
36	`(p)-[:LIKES]->(m)<-[:REPLY_OF]-(c),` `(m)-[:HAS_TAG]->(t),` `(m)-[:IS_LOCATED_IN]->(l)`	*	yes	yes
37	`(m)-[:IS_LOCATED_IN]->(l:Place),` `(m:Comment)-[:REPLY_OF]->(m2:Message),` `(m)-[:HAS_CREATOR]->(p:Person)`	*	yes	yes (redundant)
38	`(f:Forum)-[:HAS_MODERATOR]->(mod:Person),` `(f)-[:HAS_MEMBER]->(mem:Person),` `(f)-[:HAS_TAG]->(x:Tag)`	*	yes	yes (redundant)
39	`(p:Person)-[:LIKES]->(m:Message)<-[:REPLY_OF]-(c:Comment),` `(m)-[:HAS_TAG]->(t:Tag),` `(m)-[:IS_LOCATED_IN]->(l:Country)`	*	yes	yes (redundant)
40	`(c:Country:Continent)<-[:IS_LOCATED_IN]-(x1),` `(c)<-[:IS_LOCATED_IN]-(x2),` `(c)<-[:IS_LOCATED_IN]-(x4),` `(c)<-[:IS_LOCATED_IN]-(x5)`	*	yes	yes (disjoint)

Table 6.2.: The catalog of test queries for the SNB (part 2).

QueryId	Pattern	Shape	Type constraints	Label constraints

41	`(c:City:Place)<-[:IS_LOCATED_IN]-(u:University),`
	`(c)<-[:IS_LOCATED_IN]-(p:Person),`
	`(c)<-[:IS_PART_OF]-(x:Country)`
		*	yes	yes (sublabels)

42	`(p)-[:HAS_INTEREST]->(t:Tag)<-[:HAS_TAG]-(f:Forum),`
	`(p)-[:IS_LOCATED_IN]->(city),`
	`(p)-[:WORK_AT]->(company),`
	`(f)-[:HAS_MODERATOR]->(mod),`
	`(t)-[:HAS_TYPE]->(class1),`
	`(class1)-[:IS_SUBCLASS_OF]->(class2)`
		#	yes	yes

43	`(p1)-[:KNOWS]->(p2),`
	`(p1)-[:IS_LOCATED_IN]->(city:Place),`
	`(p1)-[:WORK_AT]->(company:Company),`
	`(city)-[:IS_PART_OF]->(country),`
	`(p1)<-[:HAS_MODERATOR]-(forum),`
	`(p2)-[:IS_LOCATED_IN]->(city),`
	`(p2)-[:STUDY_AT]->(university)`
		#	yes	yes

Table 6.3.: The catalog of test queries for the SNB (part 3).

6.2. Analysis

The complete results of our tests can be found in Appendix A.1.

6.2.1. Comparison of the Correlation between Actual and Estimated Result Cardinalities

For any predictive model it is desirable that its predictions are strongly linearly correlated with the actual observations. A strong linear correlation means that, the model behaves like reality (apart from a constant shift and rescaling).

For our cardinality estimations, a strong linear correlation between the actual result cardinality and the estimated cardinality implies for instance that, if query A has a smaller result cardinality than query B, the estimated cardinality of A will probably also be smaller than the estimated cardinality of B. This property is crucial for any optimizer, in order to choose the execution plan where the intermediate cardinalities are minimal.

Our hypothesis is that, our estimations are *significantly* more correlated with the actual result cardinalities than Neo4j's estimations,

In order to prove this hypothesis, we have to use statistical tests. We compute these tests in R, using the *cocor* package [1][5]. Cocor computes several tests at once, including Pearson and Filon's z-test.

In the following R code, the variables have the following meaning:

- `ResultSize` is the actual result cardinality of a query.
- `Neo4jEstSize` is Neo4j's estimate of the result cardinality of a query.
- `CascadesEstSize` is our estimate of the result cardinality of a query.

We want to show that our estimations correlate better with the actual result cardinalities than the estimations of Neo4j.

The corresponding null hypothesis is that, the correlation between `ResultSize` and `Neo4jEstSize` is greater or equal than the correlation between `ResultSize` and `CascadesEstSize`.

The following R statement runs the test:

```
cocor(~ResultSize + EstNeo4j | ResultSize + EstCascades,
      t, alternative = "less", conf.level = 0.99)
```

The test output tells us, that the actual result cardinality is only marginally correlated with the estimations of Neo4j ($r = 0.0409$), but strongly correlated with our estimations ($r = 0.8258$).

Moreover, Table 6.4 shows that all tests run by cocor tell us to reject our null hypothesis at 99% confidence level ($\alpha = 0.01$)

We can therefore conclude that our estimations are significantly stronger correlated with the actual result size than Neo4j's estimations.

The complete test output can be found in Appendix A.2.

[5]`https://cran.r-project.org/web/packages/cocor/index.html`, 18.06.2017

Table 6.4.: Formatted output of the tests run by cocor.

Test name	Test result	Null hypothesis rejected?

Pearson and Filon's z (1898)	z = -4.8927
	p-value = 0.0000
		Yes

Hotelling's t (1940)	t = -6.2367
	df = 41
	p-value = 0.0000
		Yes

Williams' t (1959)	t = -5.4285
	df = 41
	p-value = 0.0000
		Yes

Olkin's z (1967)	z = -4.8927
	p-value = 0.0000
		Yes

Dunn and Clark's z (1969)	z = -4.9999
	p-value = 0.0000
		Yes

Hendrickson, Stanley, and Hills' (1970) modification of Williams' t (1959)	t = -6.2169
	df = 41
	p-value = 0.0000
		Yes

Steiger's (1980) modification of Dunn and Clark's z (1969) using average correlations	z = -4.8416
	p-value = 0.0000
		Yes

Meng, Rosenthal, and Rubin's z (1992)	z = -4.7835
	p-value = 0.0000
		Yes

Hittner, May, and Silver's (2003) modification of Dunn and Clark's z (1969) using a backtransformed average Fisher's (1921) Z procedure	z = -4.7825
	p-value = 0.0000
		Yes

Zou's (2007) confidence interval	99% confidence interval for r.jk - r.jh: -1.1878 -0.3610	Yes

6.2.2. Comparison of the Estimation Errors

The predictions produced by a model should not only have a strong linear correlation with real observations, they should also be *close* to the observations with respect to some error metric. For our model, we use the *absolute estimation error* and the *relative estimation error* as error metrics.

Definition 6.2.1 (Absolute and relative estimation error)**.** Take a CSP query c with result cardinality x. Suppose an optimizer estimates the result cardinality of c as $\hat{x}$. The absolute error of this estimation is defined as

$$\epsilon(x, \hat{x}) := |x - \hat{x}|$$

If $x \neq 0$, the relative error of this estimation is defined as

$$\eta(x, \hat{x}) := \frac{\epsilon(x, \hat{x})}{|x|}$$

To decide whether there is a significant difference between the estimation errors of the optimizers, we use the test for paired observations described by Raj Jain [8, p. 209].

For each query of the catalog, we observe the (absolute/relative) estimation error of both optimizers. For each pair of error observations, we compute the difference. Then we compute a confidence interval for this difference. If this confidence interval includes zero, the systems are not significantly different. If it does not include zero, there is a significant difference between the (absolute/relative) estimation errors of the optimizers.

Absolute Estimation Error

Figure 6.3 shows the absolute estimation error ϵ of the two estimation models for the queries in the SNB query catalog. We transform the error scale using the inverse hyperbolic sine[6], because the error values vary by several orders of magnitude. The figure indicates that our estimations generate the smaller absolute errors.

Figure 6.4 shows the difference d_ϵ between the absolute error of Neo4j's estimations and the absolute error of our estimations. Like above, we transform the scale using the inverse hyperbolic sine. Again, we see that our estimations generate the smaller absolute errors for most of the queries in question.

Table 6.5 shows that on average, our estimations are 2 869 783 102 subgraphs closer to the actual result cardinality than the estimations of Neo4j.

In order to test whether the observed difference in absolute errors is already *significant*, we compute the 99% confidence interval of d_ϵ using the simple procedure described by Raj Jain [8, p. 209]. Let $t(0.99, n-1)$ denote the 0.99-quantile of a t-variate with $n-1$

[6]The inverse hyperbolic sine transformation (asinh) allows to cover many orders of magnitude and is defined for all real numbers. cf. `http://wresch.github.io/2013/03/08/asinh-scales-in-ggplot2.html`, 18.06.2017

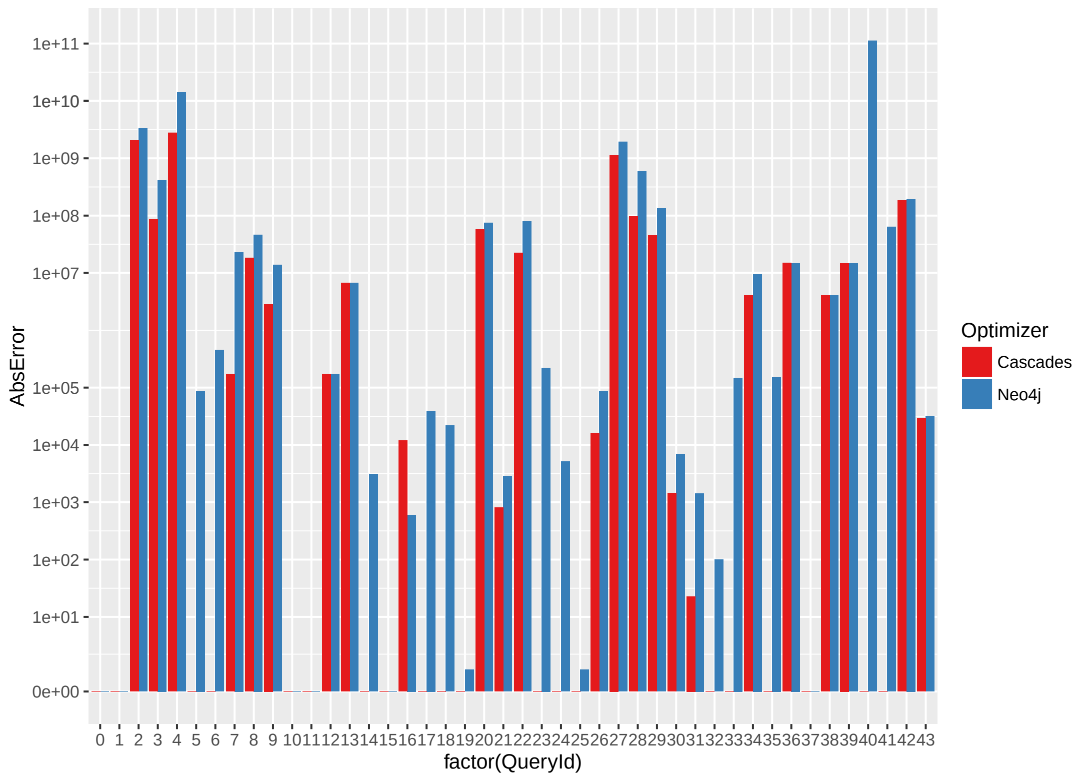


Figure 6.3.: Absolute errors of our estimations (Cascades, *red*) and Neo4j's estimations (Neo4j, *blue*) for the SNB query catalog.

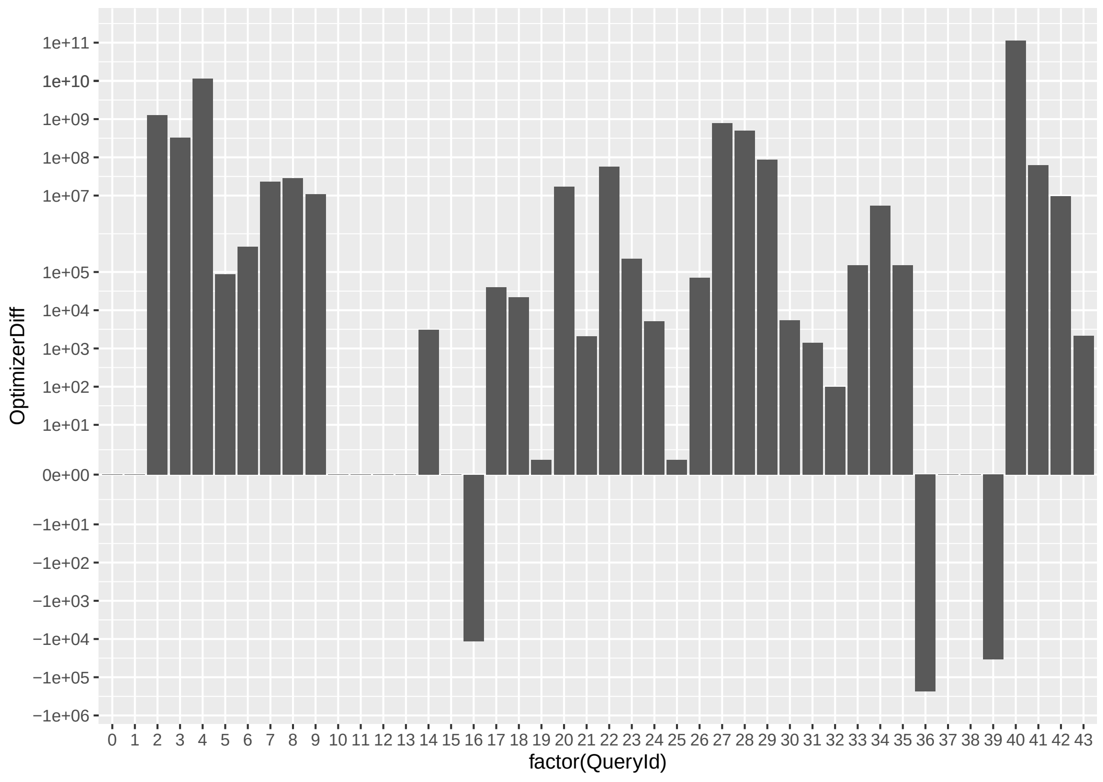


Figure 6.4.: Difference between the absolute error of Neo4j's estimations and our estimations for the SNB query catalog.

n	mean(d_ϵ)	var(d_ϵ)
44	2 869 783 102	$2.850809 \cdot 10^{20}$

Table 6.5.: Sample size, mean and variance of d_ϵ.

degrees of freedom. Then the confidence interval c_ϵ is

$$\begin{aligned} c_\epsilon &= \text{mean}(d_\epsilon) \pm t(0.99, n-1) \cdot \sqrt{\frac{\text{var}(d_\epsilon)}{n}} \\ &= 2869783102 \pm 2.41625 \cdot \sqrt{\frac{2.850809 \cdot 10^{20}}{44}} \\ &\approx (-3280563476.14, 9020129680.14) \end{aligned} \tag{6.3}$$

This confidence interval includes zero. Consequently, our current SNB query catalog is not sufficient to conclude that our estimations have significantly smaller absolute estimation errors than Neo4j's estimations.

To be able to prove this in future experiments, we should increase the sample size (i.e., the number of queries in the catalog) in order to decrease the variance of d_ϵ and thus reduce the size of the confidence interval.

Relative Estimation Error

The relative estimation error η is only defined for queries having a non-empty result. In our query catalog, these are 31 queries. Figure 6.5 shows the relative estimation error of our and Neo4j's estimation model for these queries. The figure indicates that our estimations generate the smaller relative errors.

Figure 6.6 shows the difference d_η between the relative error of Neo4j's estimations and the relative error of our estimations. Again, we see that our estimations generate the lower relative errors for most of the queries.

Table 6.6 shows that on average, our relative estimation error is 0.8096792 smaller than the relative estimation error of Neo4j.

n	mean(d_η)	var(d_η)
31	0.809679	1.661361

Table 6.6.: Sample size, mean and variance of d_η.

In order to test whether the observed difference in relative errors is already *significant*, we compute the 99% confidence interval of d_η using the same procedure as above. The

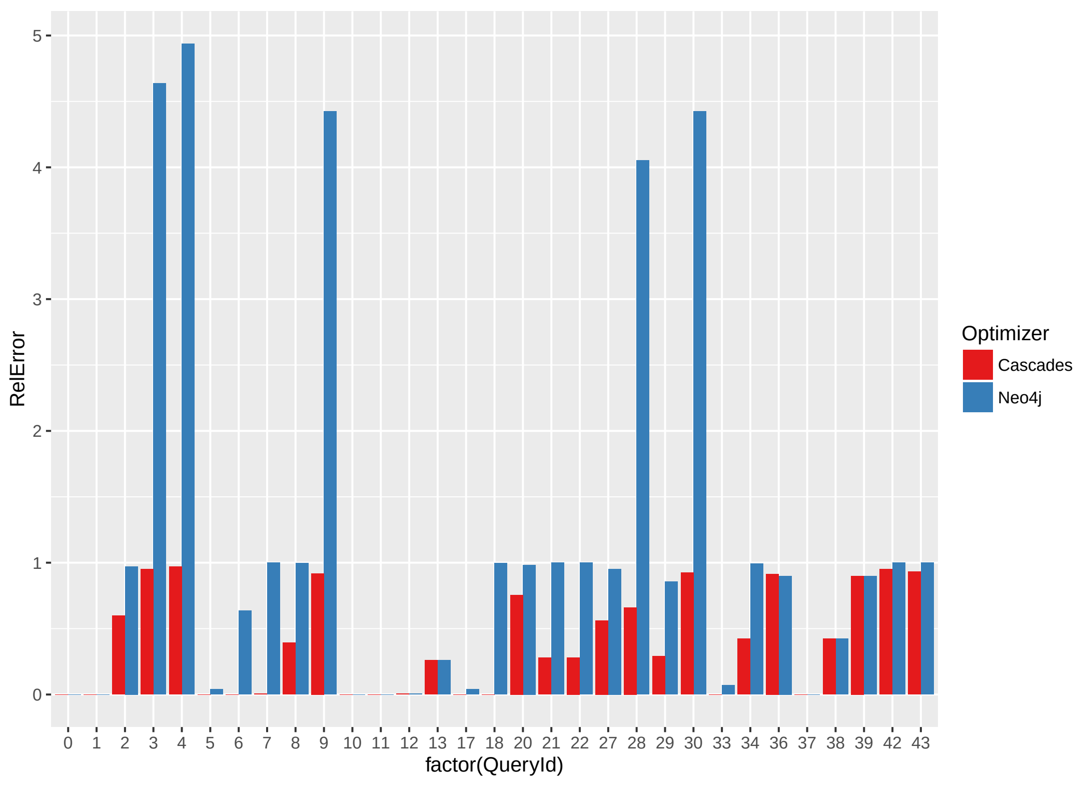


Figure 6.5.: Relative errors of our estimations (Cascades, *red*) and Neo4j's estimations (Neo4j, *blue*) for the SNB query catalog.

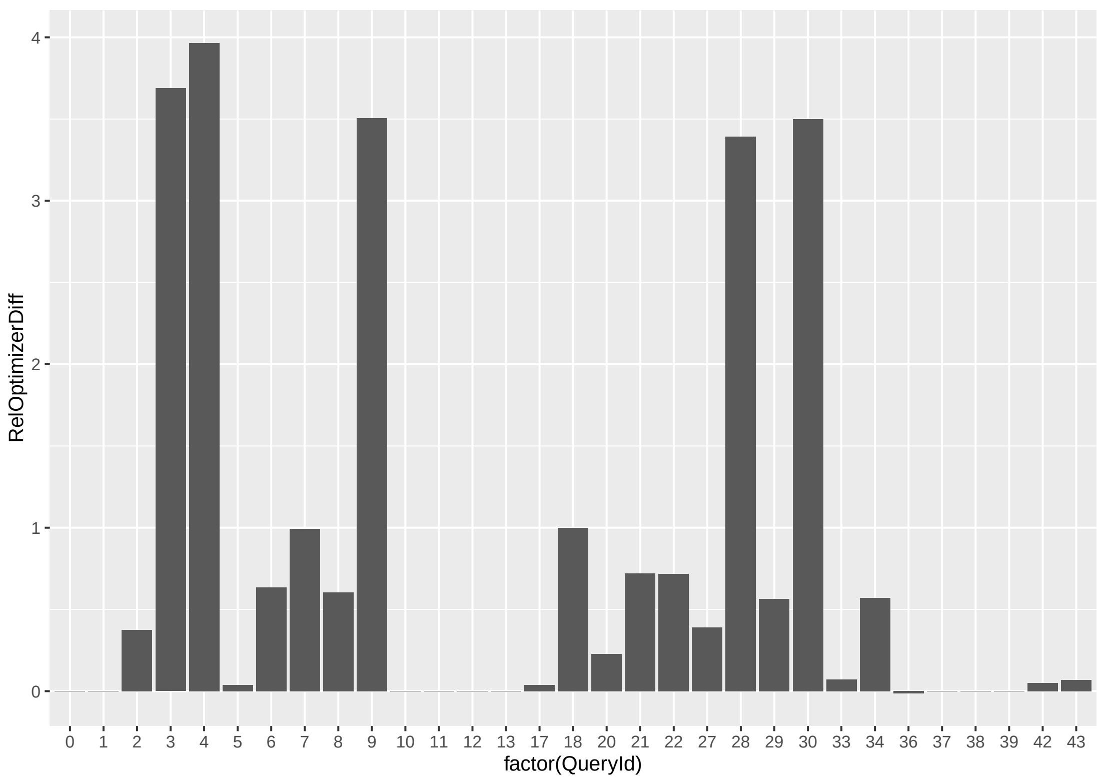


Figure 6.6.: Difference between the relative error of Neo4j's estimations and our estimations for the SNB query catalog.

confidence interval c_η is

$$\begin{aligned} c_\eta &= \text{mean}(d_\eta) \pm t(0.99, n-1) \cdot \sqrt{\frac{\text{var}(d_\eta)}{n}} \\ &= 0.809679 \pm 2.452824 \cdot \sqrt{\frac{1.661361}{31}} \\ &\approx (0.24185, 1.377507) \end{aligned} \tag{6.4}$$

This confidence interval is located above zero. Hence, we can conclude by 99% significance that our estimations generate smaller relative errors than Neo4j's estimations.

7. Summary

In this thesis, we have layed some important groundwork for sophisticated graph query optimization.

A Graph Query Algebra We have formally introduced a query algebra for graph databases, extending the algebra of Hölsch et al. [6] (cf. Chapter 2). We have proven that this graph algebra is well-defined and complete. Currently, our graph algebra allows to express subgraph matching with label, type and uniqueness constraints, which is an essential part of any graph query language.

A Statistical Model to Estimate Result Cardinality Moreover, we have illustrated that a global assumption on the distribution of node labels in a graph database can lead to severe estimation errors for datasets where this assumption is violated. We have introduced two new data structures, the *label partition* and the *sublabel map*, to address this problem.

We have then implemented our own cardinality estimation model in the Neo4j graph database. The model provides cardinality estimations for a subset of operators of our graph algebra, namely for the GetNodes, Expand and NodeLabelSelection operators (cf. Section 5.7 for the corresponding subset of Cypher). For each of these operators, we have explained in detail, which assumptions lead to the respective estimation function. Our model uses similar database statistics as Neo4j, plus a label partition and a sublabel map. Using the information provided by these two data structures, our model can avoid estimation errors if labels are not independent. Furthermore, by keeping track of the *label probabilities* at each node variable of the query pattern, our model is able to deal with strong dependencies between node labels and the presence of relationships.

We have compared our estimation model to the estimation model of Neo4j in a statistical analysis. On a catalog of 44 queries about the LDBC Social Network Benchmark (SNB) [2], our estimations are significantly more correlated with the actual result sizes than the estimations of Neo4j. Our model also has significantly lower relative estimation errors.

8. Future Work

Generalizing the Graph Algebra We believe that a general graph query algebra is a powerful mathematical tool to design new graph query optimizers and to analyze the variety of existing graph databases using a common language. This motivation has led to the first version of the graph algebra in this thesis. In the future, we want to extend our algebra to cover more use cases, e.g. result projections and shortest-paths queries.

Conducting More Tests The tests of our model for graph query cardinality estimation have revealed, that it correlates better with the actual result sizes and generates lower relative estimation errors than Neo4j's current model on a catalog of 44 queries about the SNB. To prove that our model also has the lower absolute estimation errors, we will have to increase the size of the query catalog. We will also conduct similar tests on other datasets (e.g. the Movie Database[1]) to prove the general superiority of our estimation model.

Extending the Query Space of the Estimation Model Currently, our estimation model is only defined for a very limited query space (cf. Section 5.2). In the future, we want to extend the query space covered by the model by defining estimation functions for the remaining operators TRAVERSE, DISTINCTSELECTION and NODEJOIN. The estimation function for the TRAVERSE operator requires statistics about the likeliness of cycles in the database graph. Defining and implementing these statistics is thus a necessary next step to complete our estimation model.

An Algorithm to Update the Label Partition and the Sublabel Map Online Our estimation model is informed by a label partition and a sublabel map to avoid estimation errors if labels are not independent. While we have illustrated methods to generate these data structures *offline*, it remains an open research question how they can be updated *online* in the presence of database updates. The design of an online algorithm to update the label partition and the sublabel map is needed to make our estimation model suitable for update-intensive graph database applications.

Designing the Remaining Parts of the Cascades Query Optimizer As mentioned in the Introduction, the estimation model proposed in this thesis is planned as a part of a new query optimizer for graph databases that will be implemented in the Cascades framework [3]. In the future, we want to design and implement the remaining parts of this optimizer. Firstly, we have to define *physical operators* that are able to produce the

[1] `https://neo4j.com/developer/movie-database/`

results of combinations of our logical operators in a specific graph DBMS. Secondly, we have to design a *cost model* to estimate the memory and CPU costs of these operators. Thirdly, we have to define a complete set of equivalence rules on the graph query algebra, which can then be used by the Cascades optimizer to explore the query space.

Combining Research on Graph and Relational Databases As a final remark we want to emphasize that, cardinality estimation in graph databases and relational databases poses very similar problems. We belief that, much of the research that has led to todays relational optimizers is also relevant to graph databases and should not be overlooked. As detailed above, we consider a general graph query algebra (motivated by the relational algebra) and a modular, algebraic graph query optimizer (motivated by the Cascades optimizer for relational databases) to be promising research areas for the advancement of graph databases.

Inversely, we are also convinced that insights from graph database research can help to solve current problems of relational databases. For instance, Guy Lohman states that estimating the *selectivity of join predicates* is a major issue in current relational databases[2]. A relational join can be translated to a relationship matching in a graph database and the join predicate can often be translated to a constraint on the relationship type. Translating the triple statistics maintained in graph databases (number of relationships of a particular type between nodes having particular labels) to triple statistics for relational databases (cardinality of a join between two tables using a particular predicate) could allow to use our graph query estimation model in a relational database and potentially improve the state-of-the-art of relational cardinality estimation.

We believe that there are many "low-hanging fruits" for researchers that simply require some transfer between the relational and graph database communities.

[2]`http://wp.sigmod.org/?p=1075`, 24.06.2017

A. Appendix

A.1. Estimation Results

The following Table A.1 lists the actual and estimated result cardinality of our ("Cascades") and Neo4j's ("Neo4j") estimation model for all queries of our SNB query catalog (cf. Tables 6.1, 6.2 and 6.3).

Query Id	Result cardinality	Optimizer	Estimated cardinality
0	3208020	Neo4j	3208020
0	3208020	Cascades	3208020
1	17399747	Neo4j	17399747
1	17399747	Cascades	17399747
2	3437578836	Neo4j	94373225.747972
2	3437578836	Cascades	1379934871.75512
3	89843365	Neo4j	506604613.375
3	89843365	Cascades	4412671.13905473
4	2850013610	Neo4j	16920550352.043
4	2850013610	Cascades	83257373.778174
5	2249865	Neo4j	2161631.28710232
5	2249865	Cascades	2249865
6	712049	Neo4j	1164571.92950792
6	712049	Cascades	712049
7	23072038	Neo4j	2018.73475764382
7	23072038	Cascades	23245105.2446421
8	46487188	Neo4j	86808.6352080099
8	46487188	Cascades	28152430.036393
9	3100057	Neo4j	16814174.3148668
9	3100057	Cascades	251961.265066976
10	2249865	Neo4j	2249865
10	2249865	Cascades	2249865
11	712049	Neo4j	712049
11	712049	Cascades	712049
12	23072038	Neo4j	23245105.2446421
12	23072038	Cascades	23245105.2446421
13	25694070	Neo4j	18987712.1067529

Table A.1.: Result cardinalities and estimated cardinalities for the SNB query catalog.

Query Id	Result cardinality	Optimizer	Estimated cardinality
13	25694070	Cascades	18987712.1067529
14	0	Neo4j	3093.94422728038
14	0	Cascades	0
15	0	Neo4j	0.000566237719945208
15	0	Cascades	0
16	0	Neo4j	602.044849159294
16	0	Cascades	12011.6657932243
17	1003380	Neo4j	964030.108852189
17	1003380	Cascades	1003380
18	21764	Neo4j	37.8056485189508
18	21764	Cascades	21764
19	0	Neo4j	1
19	0	Cascades	0
20	75634179	Neo4j	1226798.8804517
20	75634179	Cascades	18445055.9133887
21	2866	Neo4j	0.0503095367235865
21	2866	Cascades	2061.93904109589
22	79456454	Neo4j	0.0486164149496298
22	79456454	Cascades	57074047.1583674
23	0	Neo4j	221453.626368289
23	0	Cascades	0
24	0	Neo4j	5151.03386824272
24	0	Cascades	0
25	0	Neo4j	1
25	0	Cascades	0
26	0	Neo4j	87215.1457160492
26	0	Cascades	16080
27	2023733846	Neo4j	94373225.747972
27	2023733846	Cascades	886179714.165105
28	145759137	Neo4j	736648031.227254
28	145759137	Cascades	49336913.4067934
29	155350932	Neo4j	288450282.998198
29	155350932	Cascades	200701455.504258
30	1575	Neo4j	8542.5282650981
30	1575	Cascades	119.743150684162
31	0	Neo4j	1433.50999196308
31	0	Cascades	22.7860463258588
32	0	Neo4j	100.307118486943
32	0	Cascades	9.28403199477767e-05
33	2078830	Neo4j	1930086.37774126

Table A.1.: Result cardinalities and estimated cardinalities for the SNB query catalog.

Query Id	Result cardinality	Optimizer	Estimated cardinality
33	2078830	Cascades	2078830
34	9581547	Neo4j	52772.8208040217
34	9581547	Cascades	5532333.8472419
35	0	Neo4j	151532.622017045
35	0	Cascades	0
36	16424251	Neo4j	1635203.68710824
36	16424251	Cascades	1399019.77194174
37	2078830	Neo4j	2078830
37	2078830	Cascades	2078830
38	9581547	Neo4j	5532333.8472419
38	9581547	Cascades	5532333.8472419
39	16424251	Neo4j	1680675.39887752
39	16424251	Cascades	1646810.95801907
40	0	Neo4j	111760545789.444
40	0	Cascades	0
41	0	Neo4j	63110960
41	0	Cascades	0
42	193478450	Neo4j	0.631243322589552
42	193478450	Cascades	9571849.53401966
43	31801	Neo4j	5.45237497100945e-05
43	31801	Cascades	2173.92552767084

A.2. R Output of the Correlation Test

```
  Results of a comparison of two overlapping
  correlations based on dependent groups

Comparison between r.jk (ResultSize, EstNeo4j) = 0.0409
               and r.jh (ResultSize, EstCascades) = 0.8258
Difference: r.jk - r.jh = -0.7848
Related correlation: r.kh = -0.0374
Data: t: j = ResultSize, k = EstNeo4j, h = EstCascades
Group size: n = 44
Null hypothesis: r.jk is equal to r.jh
Alternative hypothesis: r.jk is less than r.jh (one-sided)
Alpha: 0.05

pearson1898: Pearson and Filon's z (1898)
  z = -4.8927, p-value = 0.0000
  Null hypothesis rejected
```

```
hotelling1940: Hotelling's t (1940)
  t = -6.2367, df = 41, p-value = 0.0000
  Null hypothesis rejected

williams1959: Williams' t (1959)
  t = -5.4285, df = 41, p-value = 0.0000
  Null hypothesis rejected

olkin1967: Olkin's z (1967)
  z = -4.8927, p-value = 0.0000
  Null hypothesis rejected

dunn1969: Dunn and Clark's z (1969)
  z = -4.9999, p-value = 0.0000
  Null hypothesis rejected

hendrickson1970: Hendrickson, Stanley, and Hills' (1970)
                 modification of Williams' t (1959)
  t = -6.2169, df = 41, p-value = 0.0000
  Null hypothesis rejected

steiger1980: Steiger's (1980) modification of Dunn and Clark's z
             (1969) using average correlations
  z = -4.8416, p-value = 0.0000
  Null hypothesis rejected

meng1992: Meng, Rosenthal, and Rubin's z (1992)
  z = -4.7835, p-value = 0.0000
  Null hypothesis rejected
  99% confidence interval for r.jk - r.jh: -1.7443 -0.5232
  Null hypothesis rejected (Upper boundary < 0)

hittner2003: Hittner, May, and Silver's (2003) modification of Dunn
             and Clark's z (1969) using a backtransformed average
             Fisher's (1921) Z procedure
  z = -4.7825, p-value = 0.0000
  Null hypothesis rejected

zou2007: Zou's (2007) confidence interval
  99% confidence interval for r.jk - r.jh: -1.1878 -0.3610
  Null hypothesis rejected (Upper boundary < 0)
```

The line `Alpha: 0.05` is probably a bug in cocor. It should be `Alpha: 0.01` to match the confidence level which was specified by the parameter `conf.level=0.99`.

Bibliography

[1] B. Diedenhofen and J. Musch. cocor: A Comprehensive Solution for the Statistical Comparison of Correlations. *PloS one*, 10(4):e0121945, 2015.

[2] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 619–630. ACM, 2015.

[3] G. Graefe. The Cascades Framework for Query Optimization. *Data Engineering Bulletin*, 18(3):19–29, 1995.

[4] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the Ninth International Conference on Data Engineering*, page 211. IEEE Computer Society, 1993.

[5] A. Gubichev and M. Then. Graph Pattern Matching: Do We Have to Reinvent the Wheel? In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, pages 8:1–8:7, New York, NY, USA, 2014. ACM.

[6] J. Hölsch and M. Grossniklaus. An Algebra and Equivalences to Transform Graph Patterns in Neo4j. In *EDBT/ICDT 2016 Workshops: EDBT Workshop on Querying Graph Structured Data (GraphQ)*, pages 1–2, 2016.

[7] J. Hölsch, T. Schmidt, and M. Grossniklaus. On the Performance of Analytical and Pattern Matching Graph Queries in Neo4j and a Relational Database. In *EDBT/ICDT 2017 Workshops: EDBT Workshop on Querying Graph Structured Data (GraphQ)*, 2017.

[8] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley, New York, 1991.

[9] V. Poosala and Y. E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *VLDB*, volume 97, pages 486–495, 1997.