Towards Translating Graph Query Language

Youning Xia

Master of Science

School of Informatics

University of Edinburgh

2023

Abstract

Property graph query languages such as Cypher have recently gained significant popularity due to the increasing application demands. On the other hand, logic programming languages such as Datalog have been explored as a graph query language in much earlier days than the current revival of interest in graph databases. Despite their shared use case, the connections and performance comparisons between the two have barely been investigated. To bridge the gap, in this paper, we present a full source-to-source pipeline that translates from Cypher queries to semantically equivalent and optimised Datalog programs to enable executions of generic graph queries on modern Datalog engines. Our experiments show that translated Cypher queries with optimisation evaluated on a modern Datalog engine demonstrate competitive query execution times compared to a leading industrial graph database.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Youning Xia*)

Acknowledgements

First and foremost, my deep gratitude and respect goes to my thesis advisor Amir Shaikhha for his dedicated support and guidance. Amir has always been very helpful in pointing me to interesting directions during the course of this research, which often time leads to exciting findings and new knowledge.

On a personal note, I would like to thank my parents and sister for their emotional support on my intellectual endeavours.

Contents

Chapter 1

Introduction

The last decade has seen the rise of interest in graph database [19]. This is mainly led by the trend that many modern applications require graph-structured data for analytics due to its intuitive way of modelling data. For instance, use cases of graph databases span various problem domains including fraud detection, social media, customer recommendation and drug discovery. Many popular commercial graph databases such as Neo4j, Memgraph, Amazon Neptune store graphs natively in the form of property graph [3] and supports the property graph query language Cypher [14] for expressing graph queries. Cypher and property graphs also contribute to the core components of the currently being developed standard graph database query language GQL [13].

Despite the recent development of property graph language, initial work of laying the theoretical foundation for graph databases was in fact conducted over 30 years ago which leverages a logic-based language called Datalog [7, 8, 9, 18]. In these early works, Datalog is explored as a graph query language, which attempts to represent graph nodes and edges via relations and construct graph queries through rules. What's more, Datalog itself is an interesting language and has been extensively studied in the database and programming language communities [1, 2], mostly outside the context of graph query. As a result, various powerful optimisation techniques [6, 20] have been developed and implemented in modern Datalog engines [5, 21, 16] to achieve efficient Datalog program evaluation.

Although the property graph language Cypher and logic programming language Datalog share the same use case of querying data from graphs, the two are mostly studied independently and hence their connections are not yet fully exploited. Property graph language such as Cypher, on the one hand, is a popular graph query language in the industry but only has very recent theoretical advancements [10, 12] and lacks

sufficient studies with respect to optimisation. Datalog, on the other hand, has well-established optimisation techniques but is not actively used as a general-purpose graph query language in practical applications.

In this thesis, we aim to bridge the gap between property graph query language and Datalog. In light of this, we present a systematic translation pipeline that translates Cypher queries to Datalog programs. The pipeline also includes a data model transformation step that converts the property graph to Datalog facts, over which the translated queries are evaluated. In addition, two optimisation techniques are proposed to rewrite the naively translated Datalog programs for more efficient evaluation. Our experiment results show that optimally translated Cypher queries executed on Datalog engine have significant improvement in runtime performance.

To the best of our knowledge, no prior work has compared the two in terms of execution performance, let alone investigating how to possibly leverage optimisations available in Datalog in evaluating property graph queries. The most relevant work on the interplay between the two is conducted in the context of examining the expressiveness of the graph query language with a simplified data model in terms of Datalog from a theoretical perspective [8, 9, 12].

The rest of the thesis is organised as follows. Chapter 2 provides the background information of Cypher and Datalog for the rest of the thesis. Chapter 3 presents the full translation pipeline from Cypher to Datalog, including data model transformation in Section 3.1, query translation in Section 3.2 and query optimisation in Section 3.3. Chapter 4 provides the experimental results that verify the effectiveness of our approach and Chapter 5 concludes our thesis with directions of future work.

Chapter 2

Background

This chapter provides key background information for the rest of the thesis. We start with introducing the graph query language Cypher in Section 2.1, by presenting its data model and basics of the query language through examples. We then present Datalog concepts in Section 2.2 and briefly touch upon its established optimisation techniques.

2.1 Cypher a Graph Query Language

Cypher is a declarative query language developed for extracting information from graph data. Such graph data is typically stored in a database, with its contents mapped to a data model called property graph, which incorporates various properties about nodes and edges of a graph dataset. To extract data from a property graph, Cypher allows users to specify graph patterns of their interests in a query. The graph patterns are used to intuitively describe some desirable conditions and relationships among graph elements (nodes, edges and their properties). Cypher then matches data that satisfies such patterns and returns the outputs in a table. In the following sections, we review the property graph model and give a high-level overview of key features of Cypher language.

2.1.1 Property Graph and Schema

Property Graph. Property graph is a popular data model for graph dataset and has gained adoption in many commercial database systems, such as Neo4j, Amazon Neptune and ArangoDB. As opposed to the perhaps more familiar graph from classic graph-theory literature that is often defined as pairs of nodes and edges (either directed or

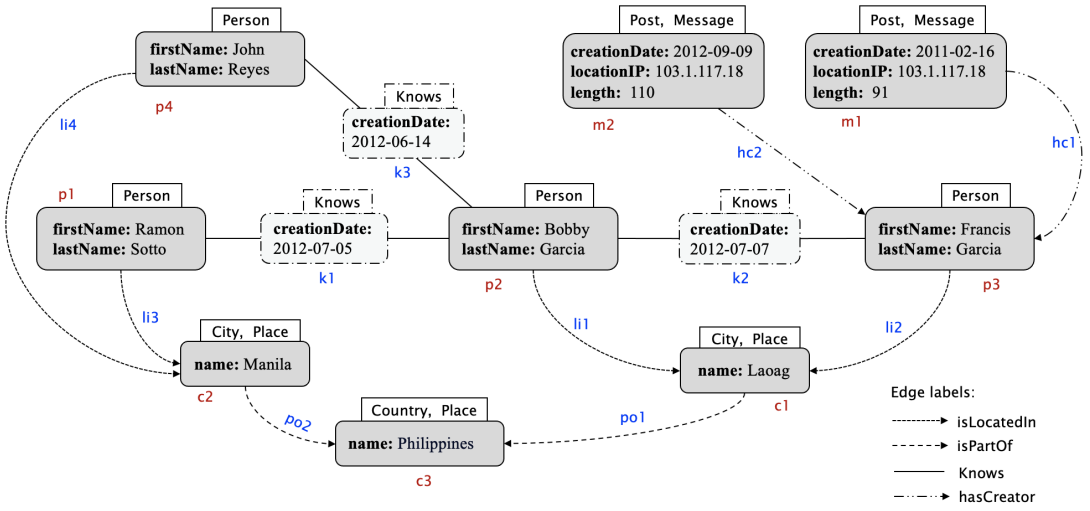

Figure 2.1: A property graph with information on people, their friends and locations. Node information is depicted in solid boxes and edge information is contained in dashed boxes. Node identifiers are coloured in red and edge identifiers in blue.

undirected), property graph is more complicated yet expressive. Property graph models the data as a partially directed (that is, both directed and undirected edges can exist in the graph) multigraph[1]. Both nodes and edges can have their own labels, and carry an arbitrary collection of key-value pairs that describe their properties. To better illustrate a property graph, we give an example graph inspired by the LDBC's Social Network Benchmark (LDBC SNB) [11]. For the formal definition of property graph, we refer the reader to Appendix A.

Example 1 (Social Network Graph)**.** Figure 2.1 shows an example property graph consisting of persons, locations and posts. Nodes and edges are identified by node identifiers (**p1**, . . . , **p4**, **c1**, **c2**, **c3**, **m1**, **m2**) and edge identifiers (**k1**, . . . , **k3**, **li1**, . . . , **li4**, **po1**, **po2**, **hc1**, **hc2**), respectively. Note that edges (**k1**, . . . , **k3**) are undirected and (**li1**, . . . , **li4**, **po1**, **po2**, **hc1**, **hc2**) are directed. Both nodes and edges can carry labels, for instance, node **p1** has a single label `Person` and edge **li1** has label `isLocatedIn`. Nodes and edges can also store a collection of key-value pairs (or interchangeably, property-value pairs). For example, node **p1** has property `firstName` and `lastName` with associated values `Ramon` and `Sotto`, respectively, and edge **k1** has property `creationDate` with value `2012-07-07`.

Property Graph Schema. Property graph schema provides a way to describe the structure of property graph data. More concretely, it specifies possible combinations of

[1] In a multigraph, there can be multiple edges between two nodes.

```
1 Schema {
2 (personType: Person {firstName:STRING, lastName:STRING}),
3 (cityType: City {name:STRING}),
4 (countryType: Country {name:STRING}),
5 (placeType: cityType | countryType),
6 (:personType)-[knowsType: Knows {creationDate:INT}]-(:personType)
7 (:personType)-[locationType: isLocatedIn]->(:placeType)
8 }
```

Figure 2.2: An example property graph schema of social media graph in Figure 2.1.

labels and properties in nodes and edges of different types, as well as constraining the types of edges allowed between certain types of nodes. As such, property graph schema plays an essential role in the query translation work of this thesis. We now provide an example of a schema for the property graph shown in Figure 2.1.

Example 2 (Social Network Graph Schema)**.** Continuing Example 1, Figure 2.2 shows an example of its schema definition. For brevity, we only show a subset of the full schema here. `Schema` specifies node types in line 2-5 and edges types in line 6-7. For example, line 2 defines a node type `personType` which has label `Person` and two property keys `firstName` and `lastName` of datatype `STRING`; line 7 defines an edge type `knowsType` whose label is `Knows` and connects nodes of type `personType`. Node and edge types can also be built from previously defined types. For instance, node type `placeType` is defined as a union of node type `cityType` and `countryType`, meaning that `placeType` can have either label `City` or `Country` and have properties associated with `City` or `Country`. Such node type is called *union type* node. Otherwise, when a node or edge type is not built from other types, it is considered as *base type*.

2.1.2 Cypher language features

We now explain how Cypher query works. As discussed previously, Cypher takes as input a property graph and matches graph patterns against the property graph to produce results. Generally, a graph pattern specifies a set of nodes and their connection via edges, with possible filtering of their labels and values of properties. Graph patterns form the core of Cypher language features and are constructed from basic building blocks, such as node, edge and path patterns. We next describe the Cypher query structure and introduce the basic building blocks of graph patterns using a simplified example query

taken from LDBC SNB.

```
1 MATCH (person:Person)-[:KNOWS*2..]-(friend),
2 (friend)-[:IS_LOCATED_IN]->(city:City)
3 WHERE person.id = "p1" AND NOT friend=person AND NOT
      (friend)-[:KNOWS]-(person)
4 WITH friend, city
5 OPTIONAL MATCH (friend)<-[:HAS_CREATOR]-(post:Post)
6 WITH friend, city, count(post) AS postCount
7 RETURN friend.firstName AS FirstName, friend.lastName AS LastName, city.name
      AS CityName, postCount
```

Figure 2.3: A Cypher query that returns information regarding names, locations and social media posts of all friends of person `p1` from the property graph in Example 2.1.

Example 3 (Recommended Posts by Friends)**.** Assume the underlying property graph defined in Example 1. The query in Figure 2.3 finds friends of friends of a given person with id `p1` and returns the name information of such friends, their location and the number of posts they have created (if any) on social media.

Cypher query is structured linearly in the sense that it is processed from the start of the query text linearly to the end. Keywords such as **`MATCH`**, **`WHERE`**, **`WITH`**, **`OPTIONAL MATCH`** and **`RETURN`** used in the query are called clauses and are used to compose the flow and intermediate data of the query in the order they appear. Each clause takes the data output by its previous clause and produces new data results to pass onto the next clause. We next analyse each clause in this example.

The **`MATCH`** clause in line 1 is used to match data of the property graph that satisfies the given graph patterns. Cypher uses "ASCII-art" syntax to specify graph patterns, where *node patterns* are enclosed in parentheses, and *edge patterns* in square brackets. Both nodes and edges may have label conditions and variables assigned to them. For example, `(city:City)` denotes a `City`-labeled node pattern with node variable `city`; node pattern `(friend)` has node variable `friend` but does not have label specified; edge pattern `-[:IS_LOCATED_IN]->` has label `IS_LOCATED_IN` but does not have an edge variable. Node and edge patterns can be chained together to form *path patterns*. Intuitively, this allows us to specify how to link some graph elements with other graph elements of our interests. For example, `(friend)-[:IS_LOCATED_IN]->(city:City)` in line 2 concatenates two node patterns with one edge pattern to form a graph path which finds the `city` that `friend` is located in.

Aside from alternating node and edge patterns to explicitly create fixed-length paths, Cypher also allows users to create variable length paths[2]. Pattern `-[:KNOWS*2..]-` in line 1 is an example of that. It indicates that two or more `KNOWS` edges should be included in this graph path. Essentially, pattern `-[:KNOWS*2..]-` is equivalent to searching pattern `-[:KNOWS]-()-[:KNOWS]-` and `-[:KNOWS]-()-[:KNOWS]-()-[:KNOWS]-` and `-[:KNOWS]-()-[:KNOWS]-...-()-...-[:KNOWS]-` so on, until it reaches a termination. A side note that `()` denotes an anonymous node.

Several path patterns can be further combined together using comma separators to form *graph patterns*. For example, path `(person:Person)-[:KNOWS*2..]-(friend)` in line 1 and path `(friend)-[:IS_LOCATED_IN]->(city:City)` in line 2 are put together to shape the graph pattern to be matched by `MATCH` clause. In this case, the two paths share the node variable `friend`, implying that the graph pattern tries to find the friends of friends of a person (length of two), or friends of friends of friends of a person (length of three), and also find the city where that friend lives. Normally in a graph pattern, multiple paths may or may not share variables; in case they do, then those paths are joined using the shared variables between them. The results produced by the **`MATCH`** clause can be found in Figure 2.4a.

The **`WHERE`** clause in line 3 applies filters to the result produced by the previous **`MATCH`** clause, restricting the values of some of its elements' properties and imposing some search conditions. These restrictions and conditions may be logically combined using **`AND`**, **`OR`**, and **`NOT`**. In our example, we are only interested in `person` with id **`p1`**; and the matched `friend` should not be the `person` itself[3]; and the `friend` should not be a direct acquaintance of the `person`. The filtered results are shown in Figure 2.4b.

The **`WITH`** clause in line 4 can be used to project only a subset of the variables in the current scope to the later part of the query and also to compute aggregation (which we will see shortly after). Here, we are only passing variable `friend` and `city` onto the next clause, which means information regarding `person` is no longer visible to the rest of the query. This gives us outputs in Figure 2.4c

The **`OPTIONAL MATCH`** clause in line 5 matches graph patterns just as **`MATCH`** does, with the difference being that if no matches are found, *null* value will be used for the missing parts of the pattern. In this case, **`OPTIONAL MATCH`** matches the `post` created by `friend`, and if `friend` does not have any post, then `post` will be set to `null`. The produced results are in Figure 2.4d.

[2]Essentially, transitive closure. We will defer the explanation to the Datalog section.

[3]This could potentially happen when there are cycles in the graph.

The **WITH** clause in line 6 is used to compute aggregation over `post`. Here, `friend` and `city` serve as the implicit grouping keys for the aggregating function **count**`(post)`. Taking the results by the previous clause shown in the above table, we count the non-null values of `post` for each (`friend`, `city`) pair and yield outputs shown in Figure 2.4e.

Finally, the last clause **RETURN** is used to project the final outputs of the query. Given the previous result, it projects values of property `firstName`, `lastName` of variable `friend`, and `name` of variable `city`, along with the value of `postCount`, which are shown in Figure 2.5.

person	friend	city
p1	p3	c1
p1	p4	c2
p3	p4	c1
p3	p1	c2
p4	p1	c2
p4	p3	c1

(a) Outputs by **MATCH** clause

person	friend	city
p1	p3	c1
p1	p4	c2

(b) Outputs by **WHERE** clause

friend	city
p3	c1
p4	c2

(c) Outputs by the first **WITH** clause

friend	city	post
p3	c1	m1
p3	c1	m2
p4	c2	null

(d) Outputs by **OPTIONAL MATCH** clause

friend	city	postCount
p3	c1	2
p4	c2	0

(e) Outputs by the second **WITH** clause

Figure 2.4: Intermediate results of the Cypher query in Example 3

FirstName	LastName	CityName	postCount
Francis	Garcia	Laoag	2
John	Reyes	Manila	0

Figure 2.5: Final query outputs by **RETURN** clause

Throughout this section, we have covered the essential language features of Cypher and briefly explained their functionalities via a running example. We refer the interested reader to [14] for more comprehensive discussions of Cypher language.

2.2 Datalog as a Query Language

Historically, Datalog was first introduced as a declarative logic programming language in the 1980s. However, it has since been actively developed by the database community as an expressive database query language due to its support of both non-recursive and recursive queries. Particularly, Datalog's abstraction for recursive computation provides a natural way to express graph queries, which we will see examples later. In this thesis, we consider an extended version of Datalog called stratified Datalog which also supports negation and aggregations[4]. Moreover, various optimisation techniques have been developed to allow more efficient evaluation of Datalog queries. As far as language goes, Datalog is very simple. In the following sections, we introduce the data model, syntax and semantics of Datalog from the view of query language and review some current work in the field of Datalog optimisations.

2.2.1 Syntax and Semantics

In this section, we give a brief primer into the syntax and semantics of Datalog. We aim to illustrate the flavour of Datalog, rather than give full formal definitions. See [1] for a thorough exposition that complements this section.

A Datalog program is typically composed of two types of relations: *extensional relations (EDB)* and *intentional relation (IDB)*. EDB can be viewed as data inputs to the Datalog program and IDB can be viewed as queries of the program. EDB and IDB can also be referred to as facts and rules, respectively. Intuitively, a Datalog program starts from EDB and derives new facts of the user's interests through IDB.

Example 4 (Locations of Friends)**.** The Datalog program in Figure 2.6 is inspired by a subset of the Cypher query from Example 3. It finds the cities where Bobby's friends live. Line 1-7 are EDB of the program. A fact `knows("Ramon", "Bobby")` can be interpreted as the first argument of relation `knows` knows the second argument, namely, Ramon knows Bobby. Similarly, `isLocatedIn("Ramon", "Manila")` implies Ramon is located in the city of Manila. Here, `knows` and `isLocatedIn` are referred to as relation names. A relation can have an arbitrary number of arguments (called arity) but the arity needs to be fixed with respect to each relation name. In this case, both relation `knows` and `isLocatedIn` have arity 2. Line 8 defines the IDB of the program. The left

[4]Datalog programs with aggregates has its own difficulties and thus has been a topic of research in various studies. However, a thorough discussion of Datalog as a language is out of the scope of our thesis, and therefore details are omitted.

```
1 knows("Ramon", "Bobby").
2 knows("Bobby", "Francis").
3 knows("Bobby", "John").
4 isLocatedIn("Ramon", "Manila").
5 isLocatedIn("Bobby", "Laoag").
6 isLocatedIn("Francis", "Laoag").
7 isLocatedIn("John", "Manila").
8 query(friend, city):- knows("Bobby", friend), isLocatedIn(friend, city).
```

Figure 2.6: A Datalog program that returns the friends of Bobby and their locations.

side of an IDB (or rule) separated by `:-`, is referred to as head and specifies the output of the IDB. The right side is referred to as body and declares how the IDB should be computed. The body of an IDB is typically formed by a sequence of relations that can be either EDB or IDB, having constants or variables in their arguments. An IDB is intended to return a set of facts that are *deducible* from its body. What that means is, for instance, with respect to the body of the IDB `query`, if `knows("Bobby", friend)` and `isLocatedIn(friend, city)` can find such values (say f_1 and c_1) for variables `friend` and `city` that the relations `knows("Bobby",`$f1$`)` and `isLocatedIn(`$f1$`,` c_1`)` exist within pre-defined facts or previously deduced rules, then we can say fact `query(`f_1`,` c_1`)` is deducible. Note that since variable `friend` is shared between the two relations, its value is shared. The following resulting facts can be obtained for this IDB.

```
query("Francis", "Laoag").
query("John", "Manila").
```

The previous example is a non-recursive query in that the rule is not defined with reference to itself. In other words, the head of the rule does not appear in its own body. What makes Datalog more interesting is that queries are allowed to be recursive, meaning they can be defined by themselves.

Example 5 (Friends of Friends)**.** The Datalog program in Figure 2.7 finds all direct and indirect acquaintances `friend` of a `person`. Line 5 defines the base case of this recursive query `query` to be the *direct* acquaintances of `person`. In line 6, `query` appears on both the head and the body of the rule. This recursion is essential for this rule to express variable length of `knows` relations included in the query, to find indirect acquaintances of `person`. By expanding this query using its base case and recursive case, the following sets of rules are implicitly implied:

```
knows("Ramon", "Bobby").
knows("Bobby", "Francis").
knows("Bobby", "John").
query(person, friend):- knows(person, friend).
query(person, friend):- query(person, someone), knows(someone, friend).
```

Figure 2.7: A transitive closure Datalog program that returns all pairs of friends

```
query(person, friend):- knows(person, friend).
query(person, friend):- knows(person, someone), knows(someone, friend).
query(person, friend):- knows(person, someone), knows(someone, someone2),
    knows(someone2, friend).
... /* the rules continue */
```

As can be seen from above, instead of having to explicitly write out all the possible cases, Datalog provides a neat abstraction to express recursion. To conclude, the following facts are deduced from the query.

```
query("Ramon", "Bobby").
query("Ramon", "Francis").
query("Ramon", "John").
query("Bobby", "Francis").
query("Bobby", "John").
```

We now relate the discussion to graph queries. In fact, the above recursive query can be viewed from a graph problem perspective. Recall the discussion from previous Cypher sections, here for example, fact `knows("Ramon", "Bobby")` can be interpreted as a directed edge where `"Ramon"` and `"Bobby"` are the source and target nodes of this edge. Rule `query(person, friend)` is trying to find all paths such that node `friend` that can be reached from node `person` via edges `knows`. This is known as *reachability* problem in graph context and the paths between `person` and `friend` are called the *transitive closure* of a graph.

In terms of Datalog query evaluations, there are several techniques that have been developed, which can typically be classified as either bottom-up evaluation or top-down evaluation strategies. We next provide a high-level overview of each to conclude this section. Bottom-up evaluation starts from the sets of facts and works towards the query. A classic technique is a fixpoint approach called *naive-evaluation* where the query evaluator starts with an empty database and repeatedly applies all rules until no new

facts are generated. Top-down evaluation starts from the query and works towards the facts instead. *SLD resolution* is a standard one of such technique in which queries are expanded and kept being evaluated until it reaches EDB facts. For more details, we refer the reader to [1].

2.2.2 Optimisation

Various optimisation techniques have been developed over the years to facilitate efficient evaluations of Datalog programs to meet the demands of industrial applications. In this section, we first present the key ideas of two popular techniques called *semi-naive evaluation* and *magic-set transformation* which are also implemented in the Datalog engine used as the backend for our work. We then conclude by providing a review of recent work in the field of Datalog optimisation.

Semi-naive evaluation. As its name suggests, semi-naive evaluation is a technique which improves the performance of naive-evaluation [20]. Since naive-evaluation progresses from facts towards rules, it is likely to constantly produce irrelevant facts that do not contribute to computing the query. Its procedure also requires to recompute all previous relations in each iteration, resulting in inefficient computation. Semi-naive evaluation solves this inefficiency by computing a safe approximation of the difference between iterations and only generates *new* facts in each iteration, bringing down the complexity of computations.

Magic-set transformation. Another well-known optimisation technique is called magic-set transformation [6]. As we mentioned above, bottom-up evaluation inevitably computes unnecessary facts that are not relevant to produce the final query output. Magic-set transformation improves it by rewriting the Datalog program with respect to its query so that irrelevant intermediate facts will not be computed. Intuitively, it tries to push some filtering operations ahead of computations, as opposed to completing all computations first and then conducting post-filtering in the naive-evaluation.

Example 6 (Magic-set Transformation)**.** Figure 2.8 shows how an example Datalog program is rewritten after applying magic-set transformation. Rather than computing the relation `friendLivesAt(person, city)` for all persons in the database first and then filtering the results for Bobby, the transformed program only computes the relation `friendLivesAt("Bobby", city)` for Bobby in the first place to avoid creating

```
1 friendLivesAt(person, city):-
      knows(person, friend),
      isLocatedIn(friend, city).
2 query(city):- friendLivesAt("Bobby",
      city).
```

(a) Original Datalog program

```
1 friendLivesAt("Bobby", city):-
      knows("Bobby", friend),
      isLocatedIn(friend, city).
2 query(city):- friendLivesAt("Bobby",
      city).
```

(b) Transformed Datalog program

Figure 2.8: Magic-set transformation is applied to a Datalog program that finds the city where Bobby's direct acquaintance lives.

unnecessary intermediate facts.

Apart from these two well-studied techniques, there are many other optimisation techniques that have been researched. *Counting* algorithm [15] is developed based on magic-set transformation which stores the number of alternative rules for deducing facts in a materialised view to minimise the number of generated facts; The *Delete and Rederive* method [15] is particularly effective for optimising recursive queries; A provenance-based incremental maintenance approach [17] is proposed to refine the deletion mechanisms of the Delete and Rederive method.

Chapter 3

From Cypher to Datalog

In this chapter, we propose an end-to-end pipeline for translating Cypher queries to semantically equivalent and optimised Datalog programs. An overview of the pipeline architecture is presented in Figure 3.1. The chapter is organised as follows. We first describe our approach for data model transformation in Section 3.1. We then explain a full naive query translation pipeline, showing how a Cypher query is decomposed and converted into Datalog IDB in Section 3.2. Finally, we present three techniques for optimising the naively-translated Datalog program to achieve more efficient query evaluation in Section 3.3.

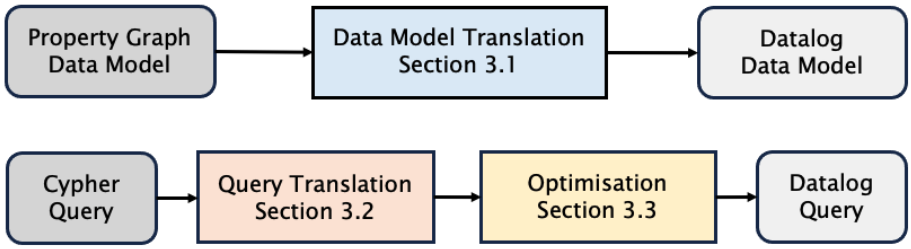

Figure 3.1: Overview of the translation pipeline architecture

3.1 Data Model Transformation

As we discussed in the previous chapter, Cypher and Datalog adopt different data models to represent data. Therefore, as the first step of query translation, we need to transform a property graph into a set of Datalog facts, over which Datalog queries can be evaluated. The transformation process is broken down into two steps. First, we map

the property graph schema to Datalog schema. Next, we map the actual property graph data to Datalog facts which conform to the mapped Datalog schema.

3.1.1 PG Schema to Datalog schema

Recall that a property graph schema consists of node and edge types. During transformation, each node and edge type is mapped to a set of Datalog relations declarations and queries (IDB) in a way that encodes label, properties information and inheritance between types. For property graph schema, we use a simplified subset of PG-schema introduced in [4]. The syntax of our schema language is given in Figure 3.2. A property graph schema S consists of a collection of elements T which are comprised of node types N and edge types E. Both types are described by their label and properties information TS, with edge types having additional label information of their source nodes and target nodes.

$$
\begin{array}{lcll}
S & ::= & \{\bar{T}\} & \textit{Schema} \\
T & ::= & N \mid E & \textit{Element type} \\
N & ::= & (TS) & \textit{Node type} \\
E & ::= & (L)-[TS]\rightarrow(L) & \textit{Edge type} \\
TS & ::= & X\ LS\ \{\overline{k:b}\} & \textit{Label property spec} \\
LS & ::= & L \mid X \mid X|X & \textit{Label spec} \\
k & ::= & k \in \mathcal{K} & \textit{Property name} \\
b & ::= & b \in \mathcal{B} & \textit{Property type}
\end{array}
$$

Figure 3.2: Grammar of property graph schema

The schema transformation rules are presented in Figure 3.3. Notation $[\![\ ]\!]_T$ is read "is transformed to", *decl* stands for Datalog relation declaration of any EDB and IDB, and *query* refers to Datalog IDB.

We now consider two concrete examples to illustrate how to apply the transformation rules to basic node and edge types.

Example 7 (Basic Node type)**.** Figure 3.4 first transforms basic node type `personType` to an EDB where node label is mapped to the relation name and the node identifier always mapped to the first argument. The ordered collection of property names with their data types is mapped to the rest of the relation arguments, with its order preserved. An IDB with node type as the relation name is then created, which is deduced from the previously defined node EDB.

$[\![\ sch\ ]\!]_T = \langle \{\ decl\ \}, \{\ query\ \} \rangle$ *Transform to a set of Datalog declarations and IDB*

$\langle d_1, q_1 \rangle \cup \langle d_2, q_2 \rangle \triangleq \langle d_1 \cup d_2, q_1 \cup q_2 \rangle$ *Union of schema elements transformation*

$[\![(x\ l\ \{\overline{k_i : b_i}\})]\!]_T = \langle \{$`.decl x(ki:bi),l(ki:bi)`$\}, \{$`.input l,x(ki):-l(ki)`$\} \rangle$

Basic node type transformation

$[\![(x_1\ x_2\ \{\overline{k_i : b_i}\})]\!]_T = \langle \{$`.decl x1(ki:bi)`$\}, \{$`x1(ki):-x2(ki)`$\} \rangle$

Inherited node type transformation

$[\![(x_1\ x_2 | x_3\ \{\overline{k_i : b_i}\})]\!]_T = \langle \{$`.decl x1(ki:bi)`$\}, \{$`x1(ki):-x2(ki),x1(ki):-x3(ki)`$\} \rangle$

Union node type transformation

$[\![(l_1) - [x\ l_3\ \{\overline{k_i\ b_i}\}] \rightarrow (l_2)]\!]_T = \langle \{$`.decl l1xl2(sid:b,tid:b,ki:bi),`
`l1l3l2(sid:b,tid:b,ki:bi)`$\}, \{$`.input l1l3l2,`
`l1xl2(sid,tid,ki):-l1l3l2(sid,tid,ki)`$\} \rangle$

Basic edge type transformation

$[\![(l_1) - [x_1\ x_2\ \{\overline{k_i : b_i}\}] \rightarrow (l_2)]\!]_T = \langle \{$`.decl l1xl2(sid:b,tid:b,ki:bi)`$\},$
$\{$`l1xl2(sid,tid,ki):-l1x2l2(sid,tid,ki)`$\} \rangle$

Inherited edge type transformation

$[\![(l_1) - [x_1\ x_2 | x_3\ \{\overline{k_i : b_i}\}] \rightarrow (l_2)]\!]_T = \langle \{$`.decl l1x1l2(sid:b,tid:b,ki:bi)`$\},$
$\{$`l1x1l2(sid,tid,ki):-l1x2l2(sid,tid,ki),l1xl2(sid,tid,ki):-l1x3l2(sid,tid,ki)`$\} \rangle$

Union edge type transformation

Figure 3.3: PG-schema transformation rules to Datalog EDB and IDB

```
1 (personType:
2   Person {
3     first:STRING,
4     last:STRING})
```

(a) PG-schema

```
.decl Person(id:UID, first:string, last:string)
.input Person
.decl PersonType(id:UID,
  first:string, last:string)
PersonType(id,fn,ln):- Person(id,fn,ln).
```

(b) Transformed Datalog schema

Figure 3.4: Transformation of basic node type `personType`

Example 8 (Basic Edge type)**.** Figure 3.5 shows the transformation of the basic edge

```
(:Person)
  -[hasInterestType:
    hasInterest]->
(:Tag)
```

(a) PG-schema

```
.decl PersonhasInterestTag(sid:UID, tid:UID)
.input PersonhasInterestTag
.decl PersonhasInterestTypeTag(sid:UID, tid:UID)
PersonhasInterestTypeTag(sid,tid):-
  PersonhasInterestTag(sid,tid).
```

(b) Transformed Datalog schema

Figure 3.5: Transformation of basic edge type `hasInterestType`

type `hasInterestType`. It is first transformed to an EDB where source node label, edge label and target node label are concatenated together to form the relation name. The first relation argument denotes the source node identifier and the second denotes the target node identifier with their data types. The rest of the relation arguments are reserved for edge properties if there are any. An IDB with edge type as the relation name is then created, which is deduced from the previously defined edge EDB.

So far we have seen the basic cases. We next present two other examples that demonstrate transforming union type node and edge to Datalog schema.

```
(cityType:
  City {name:STRING}),

(countryType:
  Country {name:STRING}),

(placeType:
  cityType |
  countryType)
```

(a) PG-schema

```
.decl cityType(id:string, name:string)
.decl City(id:string, name:string)
.input City
cityType(id, n):- City(id, n).
.decl countryType(id:string, name:string)
.decl Country(id:string, name:string)
.input Country
countryType(id, n):- Country(id, n).
.decl placeType(id:string, name:string)
placeType(id, n):- cityType(id, n).
placeType(id, n):- countryType(id, n).
```

(b) Transformed Datalog schema

Figure 3.6: Transformation of union node type `placeType`

Example 9 (Union Node type)**.** Figure 3.6 transforms union node type `placeType` to Datalog schema. Similar to Example 7, basic node types `cityType` and `countryType` are mapped 1:1 to Datalog IDB `cityType` and `countryType` which are derived from EDB `City` and `Country` respectively. Since node type `placeType` is a union of the previous

two, it is mapped to a Datalog IDB `placeType` which is defined as the union of IDB `cityType` and `countryType`.

Example 10 (Union Edge type)**.** Figure 3.7 shows the transformation of union edge type `messageHCType` to Datalog schema. Base edge types `postHCType` and `commentHCType` are mapped 1:1 to IDB `postHCType` and `commentHCType`, deriving from corresponding EDB. Union edge type `messageHCType` is mapped to an IDB which is defined as a union of IDB `postHCType` and `commentHCType`.

```
(:Post)
 -[postHCType:
     hasCreator]->(:Person),

(:Comment)
 -[commentHCType:
     hasCreator]->(:Person),

(messageHCType: postHCType |
    commentHCType)
```

(a) PG-schema

```
/* Assume EDB is defined previously */
.decl postHCType(s:string,t:string)
postHCType(s,t):- PosthasCreatorPerson(s,t).
.decl commentHCType(s:string,t:string)
commentHCType(s,t):-
    CommenthasCreatorPerson(s,t).
.decl messageHCType(s:string,t:string)
messageHCType(s,t):- postHCType(s,t).
messageHCType(s,t):- commentHCType(s,t).
```

(b) Transformed Datalog schema

Figure 3.7: Transformation of union edge type `messageHCType`

3.1.2 PG to Datalog facts

The second stage of the data model transformation process is to map property graph data to Datalog facts. This is conducted by mapping data instances of the property graph schema to Datalog facts over the corresponding Datalog schema. To illustrate, given a property graph G defined in Figure 2.1 with an example schema S specified in Figure 2.2. Example 11 and Example 12 show mappings from node and edge data instances to Datalog facts respectively.

Example 11 (Node data Transformation)**.** Figure 3.8 shows the mapping from data instances of node `Person` to Datalog EDB `Person`.

Example 12 (Edge data Transformation)**.** Figure 3.9 shows the mapping from data instances of edge `isLocatedIn` to Datalog EDB `PersonisLocatedInPlace`. Note that only the identifiers of source nodes and target nodes are encoded in the facts.

```
(:Person, {firstName:"John",
    lastName:"Reyes"}),
(:Person, {firstName:"Ramon",
    lastName:"Sotto"}),
(:Person, {firstName:"Bobby",
    lastName:"Garcia"})
```

(a) Node data instances

```
Person("p4","John", "Reyes")

Person("p1","Ramon", "Sotto")

Person("p2","Bobby", "Garcia")
```

(b) Datalog facts

Figure 3.8: Transformation of `Person` node data to Datalog facts

```
(:Person, {firstName:"John",
    lastName:"Reyes"})
 -[:isLocatedIn]->
(:City, {name:"Manila"),
(:Person, {firstName:"Ramon",
    lastName:"Sotto"})
 -[:isLocatedIn]->
(:City, {name:"Manila"),
(:Person, {firstName:"Bobby",
    lastName:"Garcia"})
 -[:isLocatedIn]->
(:City, {name:"Laoag")
```

(a) Edge data instances

```
Person("p4","John", "Reyes")
Person("p1","Ramon", "Sotto")
Person("p2","Bobby", "Garcia")
City("c1", "Laoag")
City("c2", "Manila")

PersonisLocatedInPlace("p4", "c2")

PersonisLocatedInPlace("p1", "c2")

PersonisLocatedInPlace("p2", "c1")
```

(b) Datalog facts

Figure 3.9: Transformation of `isLocatedIn` edge data to Datalog facts

Note that only Datalog EDB are mapped data instances from property graph. Facts for IDB are deduced from bodies.

3.2 Datalog Translation for Cypher Queries

We now discuss how to translate a Cypher query into a Datalog program. The translation pipeline is presented in several steps. The first step consists of decomposing a Cypher query into a sequence of query fragments according to the type of operations being performed. The remaining steps describe a systematic approach to translate each Cypher query fragment into equivalent Datalog rules and to incrementally compose these rules to produce the final Datalog query.

3.2.1 Overview

As we have seen in Chapter 2, Cypher query is a composition of linearly ordered clauses, each of which takes the output from previous clause as input and produces results that get passed onto the next clause. We develop the query translation strategy by leveraging such decomposability of clauses. To start with, a Cypher query is first decomposed by clauses such as **MATCH**, **WHERE**, **WITH** and then each clause statement is translated into Datalog rules. Due to the dependency between clauses, the translated Datalog rules will be dependent on previously defined rules. As such, the full translation is performed by incrementally constructing Datalog rules in the same order as the clause statements until reaching the end of the Cypher query. We use the same Cypher query from Example 3 as a running example to first illustrate the process of query translation on a high level.

```
1  MATCH
2   (person:Person)-[:KNOWS*2..]-(friend),
3   (friend)-[:IS_LOCATED_IN]->(city:City)
4  WHERE person.id = "p1"
5   AND NOT friend=person
6   AND NOT (friend)-[:KNOWS]-(person)
7  WITH friend, city
8  OPTIONAL MATCH
9   (friend)<-[:HAS_CREATOR]-(post:Post)
10 WITH friend, city,
11     count(post) AS postCount
12 RETURN
13  friend.firstName AS FirstName,
14  friend.lastName AS LastName,
15  city.name AS CityName,
16  postCount
```

(a) Cypher query

```
MATCH_state(...):- ...

WHERE_state(...):-
    MATCH_state(...), ...

WITH_state1(...):-
    WHERE_state(...), ...

OPTIONAL_MATCH_state(...):-
    WITH_state1(...), ...

WITH_state2(...):-
    OPTIONAL_MATCH_state(...), ...

RETURN_state(...):-
    WITH_state2(...), ...
```

(b) Overview of translated Datalog program

Figure 3.10: Cypher query from Example 3 and overview of translated Datalog program

Example 13 (Running Example)**.** Figure 3.10 shows an overview of how a Cypher query is translated to a series of Datalog IDB. The query in Figure 3.10a can be broken down into six individual clause statements: line 1-3 correspond to the first clause statement for graph pattern matching; line 4-6 are the second clause statement for filtering conditions; line 7 is the third clause statement for variable scoping; line 8-9 are the fourth clause statement for optional graph pattern matching; line 10-11 is the fifth

statement for aggregation; line 12-16 is the final statement for query result projection. Each of the statements are translated into Datalog IDB, which are used to compose the final Datalog program that has the structure shown in Figure 3.10b.

Although not shown above, some additional intermediate Datalog IDB are needed to handle cases of variable length paths (transitive closure), as we shall see shortly.

The rest of Section 3.2 describes the translation procedure in more details for the subset of Cypher clauses considered in our work.

3.2.2 Match and Graph Patterns

The core idea of translating a `MATCH` clause is to extract the elements of its matching graph patterns and use Datalog relations that stand for nodes and edges to represent the graph patterns as conjunctions of relations. Such Datalog relations are previously defined during the data model transformation process. We now demonstrate this process using the first clause statement of the running example.

```
(person:Person)
  -[:KNOWS*2..]-
(friend)
```

(a) The first path pattern

```
.decl KnowsTc(x:UID, y:UID)
KnowsTc(x, y):- Knows(x, z), Knows(z, y).
KnowsTc(x, y):- Knows(x, z), KnowsTc(z, y).
.decl path1(person:UID, friend:UID)
path1(person, friend):- KnowsTc(person, friend).
```

(b) Translated Datalog IDB

Figure 3.11: Translation of the first path pattern

```
(friend)
  -[:IS_LOCATED_IN]->
(city:City)
```

(a) The second path pattern

```
.decl path2(friend:UID, city:UID)
path2(friend, city):-
    PersonisLocatedInPlace(friend, city).
```

(b) Translated Datalog IDB

Figure 3.12: Translation of the second path pattern

Example 14 (Translate **`MATCH`** clause)**.** Continuing Example 13, Figure 3.11, 3.12, and 3.13 illustrate the step by step translation process for the first **`MATCH`** clause statement. The graph patterns in the statement can be decomposed into two graph paths where

```
MATCH
(person:Person)
  -[:KNOWS*2..]-(friend),
(friend)
  -[:IS_LOCATED_IN]->(city:City)
```

(a) Cypher clause

```
.decl MATCH_state(person:UID,friend:UID)
MATCH_state(person,friend,city):-
    path1(person,friend),
    path2(friend,city).
```

(b) Translated Datalog IDB

Figure 3.13: Translation of `MATCH` clause statement

node variable `friend` is shared between them. The first path is a variable length path with source and target node having `Person` type, connected by undirected edge `KNOWS`. To simplify the discussion, we assume the following relation `Knows` that represents the edge `KNOWS` including source and target node as a result of data model transformation. The first path which is in effect a transitive closure (as we see in Chapter 2) can be translated as `path1` shown in Figure 3.11b. The intermediate IDB `KnowsTc(x, y)` defines the transitive closure and IDB `path1` defines all possible pairs of `person` and `friend` obtainable from the path pattern. The variable arguments of IDB `path1` are determined by the variables specified for the path in the original Cypher query. Similarly, the second path can be translated into `path2` depicted in Figure 3.12b. The reason of choosing EDB `PersonisLocatedInPlace` is because an edge with label `IS_LOCATED_AT` connecting `Person`-labeled node to `City`-labeled node is mapped to `PersonisLocatedInPlace` during data model transformation.

3.2.3 Filter

`WHERE` clause is essentially filtering its input by some conditions such as constraints on label or property values. To translate a `WHERE` clause statement, we need to extract its filtering conditions and turn these into equivalent Datalog expressions.

Example 15 (Translate `WHERE` clause)**.** Continuing Example 14, Figure illustrate the translation process for the second `WHERE` clause statement. There are three conditions to filter here: (1) a constraint on the identifier value of node variable `person`, which can be translated to `Person(person, _, _), person = "p1"` where the first argument of relation `Person` denotes identifier; (2) the second condition restricts that node `friend` should be different from node `person`, which is translated to `person != friend` where `!=` denotes *not equals*; (3) the third condition is a search path constraint that there is

```
WHERE
 person.id = "p1"
 AND NOT friend=person
 AND NOT
 (friend)-[:KNOWS]-(person)
```

(a) Cypher clause

```
.decl
WHERE_state(person:UID,friend:UID,city:UID)
WHERE_state(person,friend,city):-
   MATCH_state(person,friend,city),
   Person(person, _, _), person = "p1",
   person != friend,
   !Knows(friend, person).
```

(b) Translated Datalog IDB

Figure 3.14: Translation of **WHERE** clause statement

no edge KNOWS connecting node friend and person, which is translated to a negated relation in Datalog, i.e. !Knows(friend, person). Since the **WHERE** clause takes input from the first **WHERE** clause, IDB MATCH_state should be included in the IDB body. Combining all translated condition expressions, the statement can be written as IDB WHERE_state shown in Figure 3.14b.

3.2.4 Variable Projection

The first **WITH** clause in Example 3.10a is used to project a subset of the variables currently in scope to the next clause. In the query context, this means only pairs of (friend, city) values are returned as the output of the **WITH** clause. This leads us to the following translation.

```
WITH friend, city
```

(a) Cypher clause

```
.decl WITH_state1(friend:UID,city:UID)
WITH_state1(friend,city):-WHERE_state(_,friend,city).
```

(b) Translated Datalog IDB

Figure 3.15: Translation of **WITH** clause statement

Example 16 (Translate **WITH** clause Part 1)**.** Continuing Example 15, Figure gives the translation for the third **WIHT** clause statement. Argument person in IDB WHERE_state is replaced with a wildcard since it is removed from the variable scope through the **WIHT** clause, and only variables friend and city remain as the arguments of rule WITH_state1.

Apart from variable projection, `WITH` clause is also used to compute aggregation which we will discuss its translation later in Section 3.2.6.

3.2.5 Optional Match

`OPTIONAL MATCH` clause is a variant of `MATCH` clause which matches graph patterns in the same way as `MATCH` does but returns *null* for the variables in the optional graph patterns if no data can be matched to them. This makes the translation challenging because it is generally tricky to handle *null* values in Datalog. Therefore, our strategy is to only deal with those *null* values that are actually relevant for computing the query results. This requires us to look ahead of the current statement and identify how those variables appearing in the optional graph patterns are used.

```
OPTIONAL MATCH
(friend)
  <-[:HAS_CREATOR]-
(post:Post)
```

(a) Cypher clause

```
.decl
 OPTIONAL_MATCH_state(friend:UID,city:UID,post:UID)
OPTIONAL_MATCH_state(friend,city,post):-
  WITH_state1(friend,city), post = "null".
OPTIONAL_MATCH_state(friend,city,post):-
  WITH_state1(friend,city),
  PosthasCreatorPerson(post, friend).
```

(b) Translated Datalog IDB

Figure 3.16: Translation of `OPTIONAL MATCH` clause statement

Example 17 (Translate `OPTIONAL MATCH` clause)**.** Continuing Example 16, Figure illustrate the translation process for the `OPTIONAL MATCH` clause statement. In Example 13, `post` is the new variable introduced in the `OPTIONAL MATCH` clause which holds the post created by `friend` if any. If not, `post` will have *null* value. Regarding its usage in the successive part of the query, `post` is aggregated in the `WITH` clause, and the aggregation result `postCount` is returned in the final `RETURN` clause. Since `post` is actively used as the target of aggregation later, in this case we do want to represent the *null* value. For convenience, we introduce string `"null"` as null value for variables whose data type is `string`. The `OPTIONAL MATCH` clause is then translated into shown in Figure 3.16b. Note that `post = "null"` will be used later to deal with the next aggregation statement.

Another example to deal with `OPTIONAL MATCH` is when the new variables introduced in the optional graph patterns do not contribute directly to the query result. As

an example, we consider the following Cypher query modified from LDBC SNB short query 7.

```
MATCH
 (m:Message)<-[:REPLY_OF]-(c:Comment)
 -[:HAS_CREATOR]->(p:Person)
OPTIONAL MATCH
 (m)-[:HAS_CREATOR]->(a:Person)
 -[r:KNOWS]-(p)
RETURN
 c.id AS commented,
 c.content AS commentContent,
 c.creationDate AS
     commentCreationDate,
 p.id AS replyAuthorId,
 p.firstName AS replyAuthorFirstName,
 p.lastName AS replyAuthorLastName
```

(a) Cypher clause

```
.decl
  MATCH_state(m:UID, c:UID, p:UID)
MATCH_state(m, c, p):-
  CommentreplyOfMessage(c,m),
  CommenthasCreatorPerson(c,p).
.decl
  OPTIONAL_MATCH_state(m:UID, c:UID,
      p:UID)
OPTIONAL_MATCH_state(m,c,p):-
    MATCH_state(m, c, p).
OPTIONAL_MATCH_state(m,c,p):-
   MATCH_state(m, c, p),
   MessagehasCreatorPerson(m,a),
   Knows(a,p).
/* Translation of RETURN omitted */
```

(b) Translated Datalog IDB

Figure 3.17: Translation of `OPTIONAL MATCH` clause statement

Example 18 (A different `OPTIONAL MATCH` case)**.** Figure 3.17 shows the translation of a different case of `OPTIONAL MATCH` clause, where new node variable `a` and edge variable `r` appearing in the `OPTIONAL MATCH` clause are not used in the successive query. In this case, the `OPTIONAL MATCH` clause is really just optionally providing an additional graph pattern to match variables in the current scope (which are `m` and `p`).

3.2.6 Aggregation

Aggregation in Cypher can be translated to Datalog due to the stratified aggregates available in Datalog.

Example 19 (Translate `WITH` clause Part 2)**.** Continuing Example 17, Figure 3.18 gives the translation for the fifth `WITH` clause statement. The `WITH` clause implicitly groups its input by variable `friend` and `city` and applies the aggregating function `count` to `post`. This operation can be equivalently broken down into two steps. First, assigning numeric value 1 to every post within a (`person`, `city`) group if the post is not *null* and assigning value 0 if post is *null*. This is done through an intermediate IDB `WITH_state2_inter`

```
WITH
 friend, city,
 count(post)
   AS postCount
```

(a) Cypher clause

```
.decl
  WITH_state2_inter(friend:UID,city:UID,sCount:int)
WITH_state2_inter(friend,city,sCount):-
  OPTIONAL_MATCH_state(friend,city,post),
  post = "null", singleCount = 0.
WITH_state2_inter(friend,city,sCount):-
  OPTIONAL_MATCH_state(friend,city,post),
  post != "null", singleCount = 1.
.decl
  WITH_state2(friend:UID,city:UID,postCount:int)
WITH_state2(friend,city,postCount):-
  WITH_state2_inter(friend,city,_),
  postCount = sum sCount :
     WITH_state2_inter(friend,city,sCount).
```

(b) Translated Datalog IDB

Figure 3.18: Translation of `WITH` clause statement with aggregation

in Figure 3.18b. The count is then equivalent to the sum of all assigned values of post for each group, which is handled by IDB `WITH_state2`. By converting counting into a summation problem, we effectively solve the issue of counting *null* value.

Aside from counting, Cypher also allows other types of aggregation such as maximum, minimum, average and so on. Since each of them needs different treatments during translation, for simplicity of discussion, we only describe the counting translation logic here and provide readers with a full list of Datalog translations of LDBC SNB queries online[1] for more aggregation examples.

3.2.7 Projection

Every Cypher query ends with a `RETURN` clause which projects the final results of the query. The projection typically projects certain property values of the variables in scope. Since property values are stored in the transformed Datalog EDB, translating `RETURN` clause mostly involves identifying the arguments of EDB that correspond to the node or edge properties of interests.

Example 20 (Translate `RETURN` clause)**.** Continuing Example 19, Figure gives the translation for the final `RETURN` clause statement. The `RETURN` clause returns the first

[1] https://github.com/yxia0/cypher-benchmark/tree/main/souffle-queries

name and last name of matched `friend` and the name of matched `city`, together with the `postCount` values obtained from the previous clause. Recall that `Person`-labeled and `City`-labeled relations have the following schema

```
Person(id:int, firstName:string, lastName:string)
City(id:int, name:string)
```

which are obtained during data model transformation. The arguments of the final IDB `RETURN_state` are defined as the attribute names returned by the Cypher query, and their values are obtained from the corresponding EDB.

```
RETURN
 friend.firstName
   AS FirstName,
 friend.lastName
   AS LastName,
 city.name
   AS CityName,
 postCount
```

(a) Cypher clause

```
.decl
 RETURN_state(FirstName:string, LastName:string,
     CityName:string, postCount:int)
RETURN_state(FirstName,LastName,CityName,postCount):-
  WITH_state2(friend,city,postCount),
  Person(friend, firstName, lastName),
  City(city, CityName).
```

(b) Translated Datalog IDB

Figure 3.19: Translation of **RETURN** clause statement

It is worth pointing out that **RETURN** clause also allows aggregation and computation of expressions to be performed at the projection phase. These can be easily extended in the translation process by leveraging the strategy from aggregation translation. Interested readers can find more examples in the Appendix.

3.3 Optimisation

So far we have presented a naive way to systematically translate a Cypher query into a Datalog program through a running example. While there are existing optimisation techniques such as magic-set transformation that we can leverage to efficiently evaluate the Datalog query, our proposed translation pipeline introduces more opportunities for query optimisation to effectively eliminate irrelevant relations and reduce the complexity of computation during translation. In the rest of the section, we discuss three such optimisation techniques.

3.3.1 IDB Inlining

The technique of inlining originates from compiler optimisation which tries to merge commonly shared expression in the program to avoid re-computation of the expression and also shrink the code size. Inspired by the idea of inling, we merge shared IDB occurred in the first pass of naively translated Datalog program to reduce the number of IDBs.

```
.decl KnowsTc(x:UID, y:UID)
.decl path1(person:UID, friend:UID)
.decl path2(friend:UID, city:UID)
.decl
 MATCH_state(person:UID,friend:UID)
.decl
 WHERE_state(person:UID,
    friend:UID,city:UID)
.decl
    WITH_state1(friend:UID,city:UID)
.decl
 OPTIONAL_MATCH_state(friend:UID,
    city:UID,post:UID)
.decl
 WITH_state2_inter(friend:UID,
    city:UID,sCount:int)
.decl
 WITH_state2(friend:UID,
    city:UID,postCount:int)
/* IDB definition omitted */
```

(a) Before IDB Inling

```
.decl KnowsTc(x:UID, y:UID)
.decl
 merge_state(friend:UID,city:UID)
merge_state(friend,city):-
  KnowsTc("p1", friend),
  PersonisLocatedInPlace(
     friend, city),
  friend != "p1",
  !Knows(friend, "p1").
.decl
 WITH_state2_inter(friend:UID,
   city:UID,sCount:int)
WITH_state2_inter(friend,city,sCount)
  :- merge_state(friend,city),
  post = "null", singleCount = 0.
WITH_state2_inter(friend,city,sCount)
  :- merge_state(friend,city),
   PosthasCreatorPerson(post, friend),
   sCount = 1.
```

(b) After IDB Inling

Figure 3.20: Apply IDB inlining optimisation to naively-translated Datalog program

Example 21 (IDB Inlining)**.** Figure 3.20 illustrates how a naively-translated Datalog program is rewritten after applying inlining optimisation. It can be observed that six IDB `path1`, `path2`, `MATCH_state`, `WHERE_state`, `WITH_STATE1` and `OPTIONAL_MATCH_state` are merged into a single IDB `merge_state` in the optimised Datalog program. Additionally, filtering condition `person = "p1"` is eliminated in the optimised program, and instead, the filtering value `"p1"` has replaced all occurrence of variable `person`.

3.3.2 Schema-driven Inlining

In the schema-driven inlining technique, we only consider EDB that either makes sense to compute from a schema perspective.

```
MATCH (m:Message {id:9})
RETURN
 m.creationDate
  AS date,
 coalesce(m.content,
     m.imageFile)
  AS content
```

(a) Cypher clause

```
.decl Post(id:UID, image:string,
    date:int,...,content:string)
.decl Comment(id:UID,
    date:int,...,content:string)
.decl Message(id:UID, date:int,
    content:string,...)
// EDB & arguments omitted
Message(...):- Post(...).
Message(...):- Comment(...).
.decl query(date:int, content:string)
query(date, content):-
 Post(9, _, date, _, _, _, content),
 content != "NULL".
query(date, content):-
 Comment(9, date, _, _, content),
 content != "NULL".
query(date, content):-
 Post(9, content, date, _, _, _, content1),
 content1 = "NULL".
```

(b) Translated Datalog IDB

Figure 3.21: Leverage schema-driven inlining optimisation to translate Datalog program

Example 22 (Schema-driven Inlining)**.** Let's consider the short query 4 (see Figure 3.21a) from LDBC SNB which matches any nodes with `Message` label, and returns its creation date and contents. If its content is null, then returns its image file. Note that `Message` is a union type of post and comment, and whilst both post and comment has property `content`, *only* post has `imageFile` property. Therefore, instead of using IDB `Message`, which matches with what the query explicitly asks for, we inline its body which are EDB `Post` and `Comment` in the translated program. Particularly, we eliminate the need to use EDB `Comment` when translating `coalesce(_, m.imageFile)`, because EDB `Comment` does not even have property `imageFile`.

3.3.3 Join Ordering

In Datalog, a conjunction of two relations `A(...), B(...)` is evaluated as the cross-product between the two. This can lead to computational inefficiency during query evaluation as the effort of computing and storing the results is quadratic in the size of `A` and `B`. Therefore, we want to avoid having translated Datalog rules that involve cross-products among multiple *large* relations. This can be done by pre-computing some cross-products and storing them as IDB for later use in the program, which is the core idea of the join ordering technique.

Deciding which cross-products in the naively-translated Datalog query to pre-compute requires the profiling information regarding the tuple size of Datalog EDB generated during the data model transformation stage. Therefore, join ordering optimisation is performed in two steps. First, we identify the cross products between Datalog EDB of large size in the translated queries. Then we add new IDB for the cross-products and replace the previous occurrence of cross-products with newly created IDB.

```
.decl
 state3(forum:int,
  post:int,postCount:int)
state3(forum:int, post:int,
    postCount:int):-
 state2(forum,_), post = 0,
     postCount = 0.
state3(forum:int, post:int,
    postCount:int):-
 state2(forum,person2),
 PosthasCreatorPerson(post,
     person2),
 ForumcontainerOfPost(forum,
     post), postCount = 1.
```

(a) Before join-reordering

```
.decl
 preJoin(forum:UID,post:UID,person2:UID)
preJoin(forum,post,person2):-
 ForumcontainerOfPost(forum, post),
     PosthasCreatorPerson(post, person2).
.decl
 state3(forum:int,post:int,postCount:int)
state3(forum:int,post:int,postCount:int):-
 state2(forum,_), post = 0, postCount = 0.
state3(forum:int,post:int,postCount:int):-
 state2(forum,person2),
 preJoin(forum,post,person2), postCount = 1.
```

(b) After join-reordering

Figure 3.22: Apply join-reordering optimisation to naively-translated Datalog program

Example 23. Consider the Datalog query fragment translated from complex query 5 of LDBC SNB (see Figure 3.22a). In the naively-translated program, IDB `state3` is deduced from EDB `PosthasCreatorPerson` and `ForumcontainerOfPost` which have tuple size of 1M respectively, and previously defined IDB `state2` which has tuple size

859K. Therefore, the computation complexity of the cross-products is $O(1M \times 1M \times 859K)$.

Since EDB `PosthasCreatorPerson` and `ForumcontainerOfPost` are joined over variable `post`, we pre-calculate this cross-product beforehand in a new IDB `preJoin` defined in Figure 3.22b and replace the cross-product in `state3` with this new IDB in the optimised version. IDB `preJoin` has computation complexity $O(1M \times 1M)$ with a resulting tuple size of 1M, and hence the cross-product complexity of `state3` is reduced to $O(859K \times 1M)$. The total complexity is therefore improved roughly from $O(1M^3)$ to $O(1M^2)$.

Chapter 4

Performance Study

In this chapter, we present an experimental study to evaluate the performance of translated Cypher queries executed on Datalog engines and verify the effectiveness of our optimisation techniques.

4.1 Setup

Systems All experiments are performed on a single machine with two AMD EPYC-7302 @ 3GHz CPUs and 503 GB of RAM running Ubuntu 20.04. We only use one logical core. All Cypher queries are evaluated on Neo4j community version 5.10.0 and Datalog programs on Souffé version 2.4. Neo4j is assigned one logical CPU core upon initialisation and is used with the default setting. To execute queries on Neo4j, we use Neo4j Python Driver version 5.11. Cypher query runtime is calculated as the sum of values *result_available_after* and *result_available_after* reported by the driver. Datalog program is executed using Soufflé's compiler mode with a single thread. A profile log is also generated for each run of Datalog programs by enabling *-p* flag. Soufflé is an in-memory system and loads all input data into memory. We calculate the runtime of Datalog program as the sum of execution time of all relations, excluding input relation data loadtime, which are all recorded in the profile log. Soufflé compilation time is also excluded from runtime since query planning time is not our main focus. To make the result comparable, for Neo4j, we conduct a warm-up running for each query so that disk IO time is excluded and the query plan is cached. All reported query runtimes are averages of five successive runs, and for Neo4j the five runs start after a warm-up running.

Dataset To gauge the potential of utilising Datalog engines as generic graph query

execution systems on real-world applications, we use LDBC Social Network Benchmark (LDBC SNB) [11], which is a commonly used graph benchmark that models a complicated social networking application with users, forums, posts and comments. The LDBC SNB uses a graph schema with 14 node types connected by 20 edge types. We generate the dataset using scale factor 10, containing a total of 3.18M nodes and 17.44M relationships. We implemented the data model transformation pipeline[1] to automate the generation of Datalog facts of the LDBS dataset.

Queries For queries, we use the LDBC SNB interactive workload [11] which consists of a set of 7 short and 12 complex read-only Cypher queries that touch a significant amount of data. Since Soufflé does not support several query features, such as ordering, extracting years and months from datetime object, limiting the number of outputs etc, we slightly modified the benchmark. We removed `ORDER BY` and `LIMIT` clauses from the queries, and also deleted filtering constraints that involve extracting year, month or day from a datetime-type attribute. In addition, since Datalog adopts set semantics, we change the `RETURN` clause in all Cypher queries to `RETURN DISTINCT` so that query results are given using set semantics. A full list of modified LDBC Cypher queries with their Datalog translation is available online[2].

4.2 End-To-End Benchmarks

In the first experiment, we compare the performance of (1) original Cypher queries (2) translated Datalog queries without optimisations (3) translated Datalog queries with the best optimisation strategy. We first manually translate the Cypher queries using our propose translation pipeline without optimisation, and then apply our optimisation techniques together with the magic-set optimisation provided by Soufflé to obtain the most performant optimisation combinations. Figure 4.1 and 4.2 show the query runtimes. Table 4.1 and 4.2 summarise the optimisation strategy adopted for each query.

For short queries, 5 out of 7 translated queries outperform Neo4j without optimisation. After applying optimisation, Query 3 significantly outperforms and Query 2 shows comparable performance to Neo4j.

For complex queries, 10 out of 12 translated queries with optimisation outperform Neo4j by a large margin. There are two queries (Q1 and Q9) perform worse than Neo4j with optimisation enabled. Both queries involve cross-products computation

[1] https://github.com/yxia0/kaeru
[2] https://github.com/yxia0/cypher-benchmark

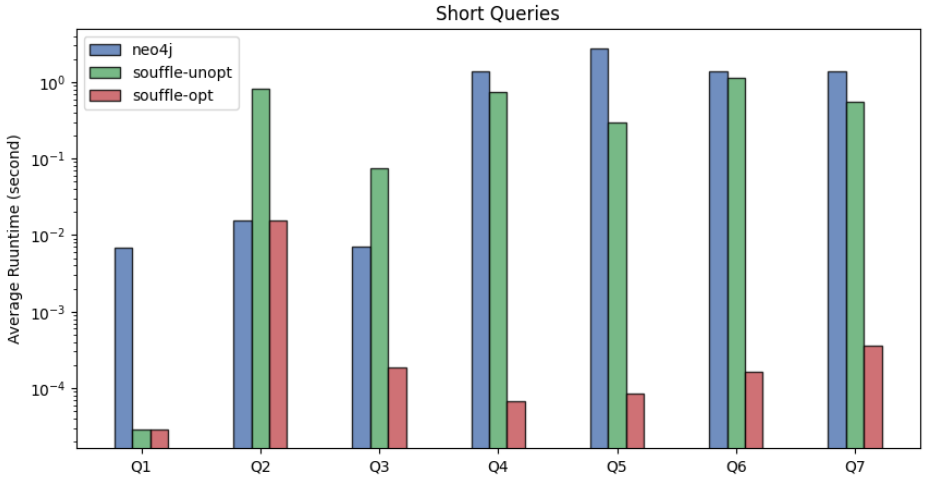

Figure 4.1: Runtimes of LDBC short queries

Opt	Q1	Q2	Q3	Q4	Q5	Q6	Q7
inline		✓	✓	✓	✓	✓	✓
join							
magic-set							

Table 4.1: Optimisation strategy for short queries

with variable length graph paths consisting of undirected edge *knows*, which increases the complexity. This also reveals the shortcomings of the underlying Datalog engine Soufflé regarding its limited capability of join optimiser.

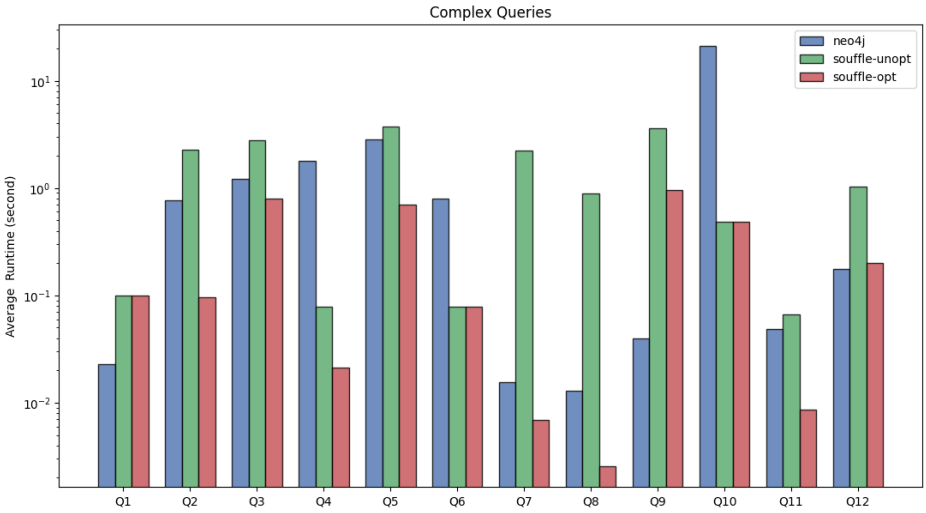

Figure 4.2: Runtimes of LDBC complex queries

We also observe that enabling magic-set transformation setting together with our

Opt	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
inline	✓	✓	✓	✓	✓		✓	✓	✓			✓
join			✓		✓				✓			✓
magic-set							✓	*			✓	

Table 4.2: Optimisation strategy for complex queries. For Query 8, applying inlining or magic-set optimisation achieves the same best query performance.

proposed query optimisation does not always lead to the best performance. Instead, it can incur some computation overheads that worsen the query evaluation time.

4.3 Evaluation of Optimisation

We next evaluate the effectiveness of individual query optimisation techniques. We take those queries of which the naively translated Datalog version performed worse than Neo4j and run experiments with their partially optimised variants where only one optimisation technique is applied at a time. The rest of the section focuses on analysing the impact of each optimisation approach.

4.3.1 Inlining

Figure 4.3 compares the performance of translated Datalog queries with and without inlining optimisation. All query performance get improved with some (SQ2, SQ3, CQ2, CQ7, CQ8) showing significant decrease in runtimes. In these 5 cases, the queries all involve some kinds of graph pattern matching with respect to *Message*-type node, which consists of *Post* and *Comment* nodes and some relationships related to them. These node and edge types have a large number of data instances and hence provide opportunities for inlining to reduce the computation of irrelevant Datalog EDB.

4.3.2 Join Ordering

Figure 4.4 shows the runtimes of translated queries with and without join reordering optimisation enabled. We observe that all queries demonstrate performance improvement. In these four cases, pre-computed cross-products typically involve relations *MessagehasCreatorPerson*, *MessageisLocatedInPlace*, *CommentreplyOfPost* which have dominant number of data tuples processed in the queries. Join ordering technique

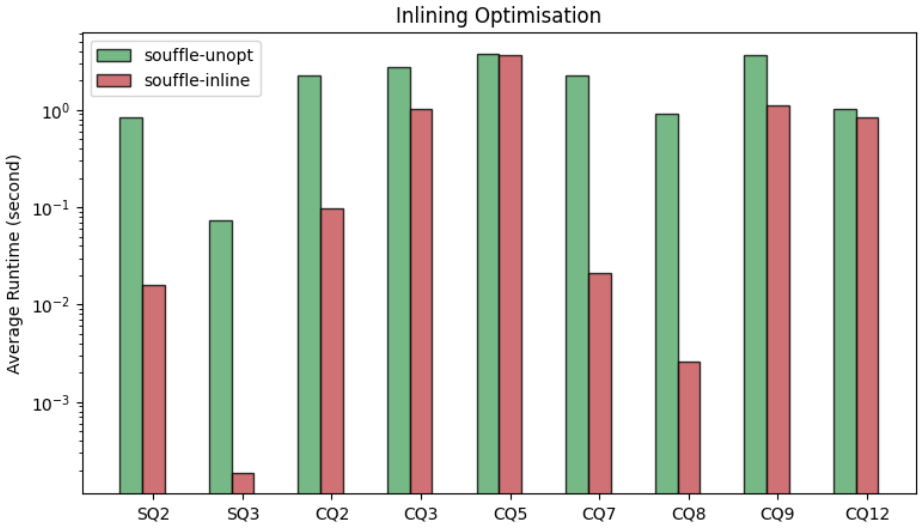

Figure 4.3: Runtime comparison with Inlining

allows the reuse of such cross-products for later computation in the queries and thus reduces computation complexity.

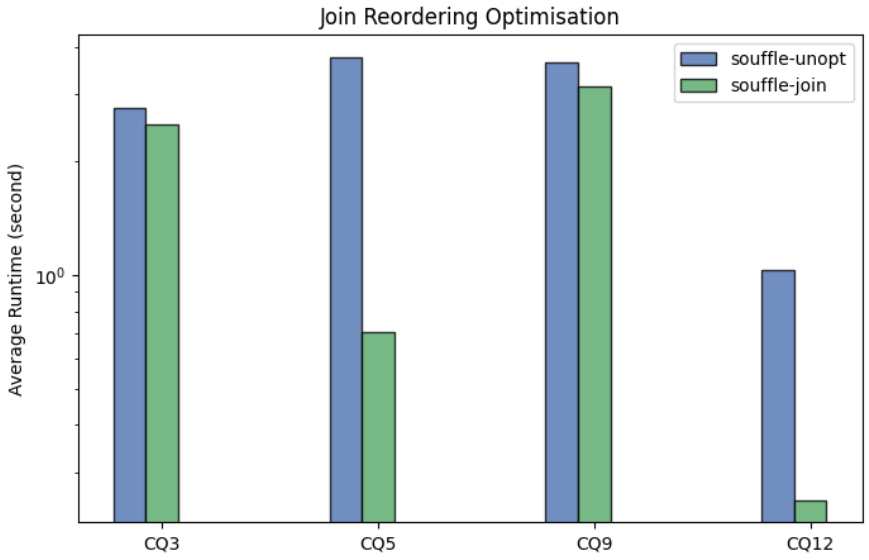

Figure 4.4: Runtime comparison with Join Reordering

It is also worth pointing out that, as can be shown in Table 4.2, join reordering is observed to be more effective when combined with inlining. This is because inlining often involves substituting some large IDB relations with only their relevant defining components (IDB bodies), which opens up opportunities for more fine-tuned join reordering.

4.3.3 Magic-set Transformation

Magic-set transformation is an optimisation technique developed by the Datalog community (as we have discussed in Chapter 2). In this experiment, we rely on Soufflé to apply magic-set transformation to the translated Datalog queries and the runtime comparison is given in Figure 4.5.

In general, magic-set transformation seems to improve query performance. There are 4 exceptions (CQ1, CQ3, CQ5, CQ12) here where the runtime becomes worse with optimisation. This requires investigation of how Soufflé transforms the queries, which is an interesting topic for our future work.

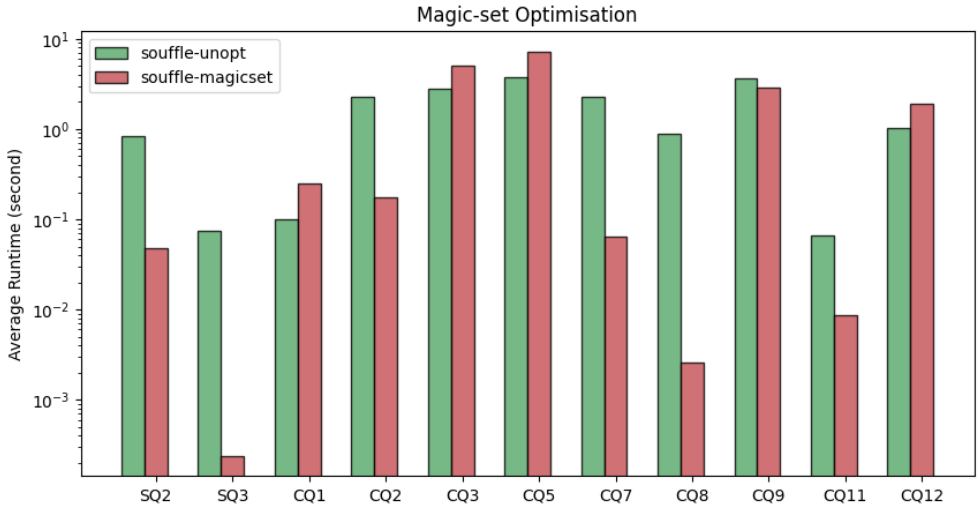

Figure 4.5: Runtime comparison with Magic-set Transformation

Chapter 5

Conclusions

In this thesis, we have presented a source-to-source translation pipeline from Cypher to Datalog, which includes data model transformation, query translation and query optimisation. The novelty of our approach consists of (1) formalisation of the schema transformation from the data model of property graph to the data model of Datalog (2) a comprehensive systematic translation process that supports core components of Cypher query, including optional graph pattern matching, transitive closure and aggregation (3) effective optimisation techniques of rewriting naively-translated Datalog program towards more efficient evaluation. Our experimental results and analysis show the performance advantage gained by executing translated Cypher queries with optimisation on modern Datalog engine.

Future work should cover several directions: First, we plan to implement the full translation pipeline as a compiler. This would enable users to easily translate Cypher queries into Datalog programs and open up opportunities for more research into the connections between the two. Second, a formalisation of the query translation rules should be provided to assist further work on proving the correctness of the translated Datalog program from a semantics equivalence perspective. Third, we can extend the translation source language from Cypher to the standard Graph Query Language, which is heavily inspired by Cypher. This will provide new opportunities for the community to improve the standardisation of graph query language through potential insights from Datalog.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Mario Alviano, Wolfgang Faber, Nicola Leone, Simona Perri, Gerald Pfeifer, and Giorgio Terracina. The disjunctive datalog system dlv. In *Datalog*, 2010.

[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. 50(5), sep 2017.

[4] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. Pg-schema: Schemas for property graphs. *Proc. ACM Manag. Data*, 1(2), jun 2023.

[5] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1371–1382, New York, NY, USA, 2015. Association for Computing Machinery.

[6] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, page 1–15, New York, NY, USA, 1985. Association for Computing Machinery.

[7] Mariano P. Consens and Alberto O. Mendelzon. Graphlog: a visual formalism for real life recursion. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1990.

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

[9] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, page 323–330, New York, NY, USA, 1987. Association for Computing Machinery.

[10] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. Graph pattern matching in gql and sql/pgq. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2246–2258, New York, NY, USA, 2022. Association for Computing Machinery.

[11] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 619–630, New York, NY, USA, 2015. Association for Computing Machinery.

[12] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. Gpc: A pattern calculus for property graphs. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '23, page 241–250, New York, NY, USA, 2023. Association for Computing Machinery.

[13] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. A researcher's digest of gql. *Proceedings of the 26th International Conference on Database Theory (ICDT 2023)*, 255:1–22, 2023.

[14] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. *In SIGMOD Conference*, page 1433–1445, 2018.

[15] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 157–166, New York, NY, USA, 1993. Association for Computing Machinery.

[16] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

[17] Mengmeng Liu, Nicholas E. Taylor, Wenchao Zhou, Zachary G. Ives, and Boon Thau Loo. Recursive computation of regions and connectivity in networks. In Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng, editors, *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 1108–1119. IEEE Computer Society, 2009.

[18] J. Paredaens, P. Peelman, and L. Tanca. G-log: a graph-based query language. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):436–453, 1995.

[19] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei R. Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. The future is big graphs: A community view on graph processing systems. *Commun. ACM*, 64(9):62–71, aug 2021.

[20] Jeffrey D. Ullman. Bottom-up beats top-down for datalog. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1989.

[21] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *SIGPLAN Not.*, 39(6):131–144, jun 2004.

Appendix A

Property Graph

We provide a formal definition of a property graph. Let $\mathcal{L}, \mathcal{P}, \mathcal{V}$ be countable sets of labels, property names and property values, respectively. A property graph is defined as a tuple $G = (N, E_d, E_u, \lambda, endpoints, src, tgt, \delta)$ where

- N is a finite set of node identifiers
- E_d as a finite set of directed edges identifiers
- E_u as a finite set of undirected edge identifiers used in G
- $\lambda : N \cup E_d \cup E_u \to 2^{\mathcal{L}}$ is a labelling function mapping node and edge identifiers to sets of labels.
- $src, tgt : E_d \to N$ define source and target node of a directed edge.
- $endpoint : E_u \to 2^N$ define endpoint nodes of an undirected edge.
- $\delta : (N \cup E_d \cup E_u) \times \mathcal{P} \to \mathcal{V}$ is a partial function mapping a (node or edge) identifier and property names to property values.