Oliver Jack Crawford

Translating Cypher: an investigation in applying graph queries to relational databases

Computer Science Tripos – Part II

Trinity Hall

12 May 2017

Proforma

Name:	**Oliver Jack Crawford**
College:	**Trinity Hall**
Project Title:	**Translating Cypher: an investigation in applying graph queries to relational databases**
Examination:	**Computer Science Tripos – Part II, June 2017**
Word Count:	**11997**[1]
Project Originator:	Dr Ripduman Sohan
Supervisor:	Mr Lucian Carata

Original Aims of the Project

The original aim of this project was to build a tool which could convert the graph database Neo4j and its graph query language Cypher, to a relational database schema and SQL respectively. An investigation of the performance and scalability of Neo4j and the Postgres RDBMS[2] would then be quantitatively evaluated. The tool, built in Java, will be tested and measured on both the Computer Lab's OPUS database, as well as other artificially created graphs. Extensions included studying possible query/database improvements, and extending the Cypher language beyond its current capabilities.

Work Completed

I have implemented a five-step working toolchain in Java that can convert graph schemas from Neo4j to a relational database (Postgres in this case). Cypher queries can then be translated to SQL. The tool can validate the results, and record the execution times on both database engines, helping me to successfully perform the evaluation for this project.

I also implemented an extension to the Cypher language. The 'iterative match' query is not currently possible on Neo4j, however, I have defined a possible syntax and semantics of such an extension, which can be translated to executable SQL.

[1] This word count was computed by `texcount -total diss.tex`.

[2] Relational Database Management System.

Special Difficulties

None.

Declaration

I, Oliver Jack Crawford of Trinity Hall, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date: 12th May 2017

Contents

List of Figures

Acknowledgements

I would like to thank my supervisor, Lucian Carata, for all his guidance in introducing me to Neo4j and for configuring the server in the Computer Lab for me. This machine proved extremely beneficial personally, as it allowed me to run tests that were very computationally expensive, whilst keeping my work machine free to look into extensions and any bug fixes.

I would also like to thank multiple people for proofreading my initial drafts and providing helpful feedback. They include my Director of Studies, Prof. Simon Moore, my parents Anne and Ian, my brother Toby, and my friends from college.

Chapter 1

Introduction

This dissertation is investigating one of the more popular strands of NoSQL: graph databases. This technology has seen substantial growth in popularity since 2013, with both organisations and researchers beginning to explore their potential. Reasons for such large growth in recent years has been due to the increasing connectivity of data, the improved performance of queries running on highly connected data, and the fact that more companies especially are realising that the semantics of a graph model fit more closely to the data they hold [1].

Understanding the performance and scalability characteristics of graph databases in comparison to relational databases is the main goal of this work. Using an automatic translation tool to convert graph schemas and graph query language to a relational database format, I aim to evaluate quantitatively where exactly they excel, and where any shortcomings may be present.

1.1 Background of project

In the last seven years, the world of database technologies has gone through an incredible transformation. With the concepts of Web 2.0 and "Big Data" becoming more popular, so is the use of new technologies, in particular NoSQL database systems.

Figure 1.1 illustrates just how popular graph databases have become in the last three years. Their popularity has increased five-fold in this short space of time, whereas RDBMSes have seen no increase at all.

But despite the acclaim and extreme amount of NoSQL databases available (currently more than 225 in use [2]), relational databases remain by far the most used in systems. This might seem surprising, given that a vast majority of the new databases aim to solve pitfalls of this technology. But relational databases are mature, sophisticated pieces of engineering that are highly compatible with software programs, and this is why they remain so well used.

My project is looking at how these two very different technologies compare when querying identical sets of hierarchical data, particularly as their size and complexity increases. For this project, I am looking specifically at Neo4j, the world's most popular

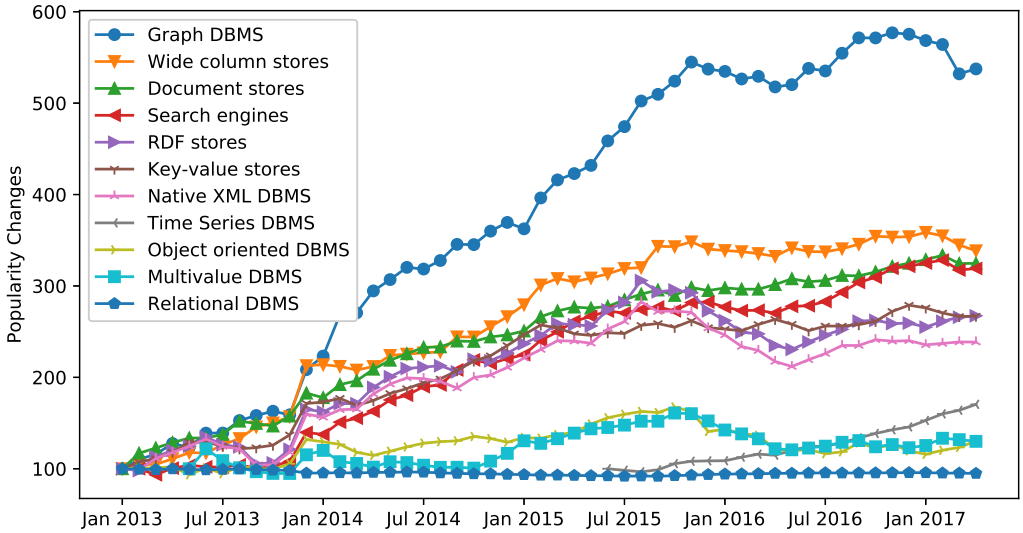

Figure 1.1: Popularity changes of various database system categories between January 2013 and April 2017. The ranking at each month is for the three most popular database systems per category. For comparison purposes, the initial value is normalised to 100. Source: `db-engines.com/en/ranking_categories`.

graph database [3], and PostgreSQL (Postgres), one of the most advanced open-source databases available.

The project will rely primarily on the tool I plan to build, which will have the ability to accept Neo4j graph databases and Neo4j's graph query language, Cypher, and automatically convert them both to an executable relational form, allowing it to be timed and evaluated against the original graph representation.

Executing this tool over a range of queries and graph structures will enable me to present useful insights into how the two databases compare. The tool itself could even have the potential to be extremely helpful for applications depending heavily on the graph database model. The initial hypothesis is that Neo4j will not necessarily scale as well as Postgres for larger graphs and certain graph queries. If this turns out to be true for a large portion of configurations, there is real value in this tool being able to compliment Neo4j in these scenarios.

Further to this work, I aim to use the tool as a prototyping testbed for extensions to the Cypher language, beyond its current capabilities, based on the translation model I plan to develop.

1.2 Motivation

The rationale for undertaking this project is to investigate how graph databases can perform and scale, and how well they compare to the tried and tested relational database format. To match the trend of "big data", the project will study closely how the performance scales for increasingly large and connected datasets.

As mentioned, I have chosen to evaluate the Neo4j database – it is an open-source, ACID-compliant transactional database with **native graph storage and processing**. Walmart, eBay, and UBS are just some of the large businesses using graph databases in their operations [4].

I hope that this project will also have applications beyond this initial work. The OPUS tool in the Computer Lab uses a sizeable and highly connected Neo4j graph database as part of their research – optimising this, or at least detecting the types of workloads Neo4j is faster or slower, will provide great insight. A result of this is that the tool should be as extensible, general, and clear as possible for future use.

1.3 Related work

To my understanding, there have been no previous attempts to convert graph database schemas and Cypher to a relational schema and SQL respectively. Most of the work in the field has been looking at the inverse problem of converting SQL and traditional databases to graphs. This is why I believe this project to be very interesting. Nonetheless, some previous related work has provided a useful starting point for my project and a point of discussion.

1.3.1 Hölsch and Grossniklaus

This 2016 paper investigated how to potentially manipulate graph queries to a relational algebra for Neo4j [5]. They mention how the technological advances over three decades of research in query optimisers for relational database systems have not filtered through to graph databases yet.

Hölsch and Grossniklaus describe in some detail a possible data model and relational algebra that Cypher and Neo4j could use. This provided a good starting point for how I planned on translating the queries myself. Firstly, it helped me consider the ideal intermediate model to represent graphs with. Secondly, it made me more aware of how to translate Cypher to SQL, once I have parsed the Cypher query and built up my own internal representation of it. The way in which a query is written can have drastic effects on the number of database hits, and thus performance, as highlighted in this paper.

1.3.2 Performance of graph databases

Performance evaluations of graph database technologies have also been done before. The company behind Neo4j naturally support their own product, and they make the following claims for why Neo4j should be preferred by developers when the data being used is both flexible and relationship orientated [6]:

- Modelling data with a high number of relationships is more efficient in Neo4j.
- Relational databases have a fixed schema, and an increasing number of joins required to query highly interconnected data will just lead to an exponential increase in both the complexity of the SQL statement, and the computing resources required.

- Developer productivity also suffers because the tabular data model is complex, harder to understand, and does not match the developer's mental model of the application. This is referred to as the *'Object Relational Impedance Mismatch'* [7].

Cypher and Neo4j may help model data which is inherently graph orientated, but unless Neo4j is scaled up across multiple database servers, it is possible that it will be slower relative to relational technology.

An independent performance evaluation of open source graph databases has also been carried out before, looking specifically at how different graph algorithms perform on various graph database engines (including Neo4j) [8].

I seek to clarify in this dissertation the extent to which the claims made in the first two items mentioned above are true. The last item is more a qualitative measurement, that is likely to be dependent on many other factors other than performance and scalability.

1.4 Outline of dissertation

This dissertation aims to fully describe and evaluate the tool developed for translating Cypher queries into SQL ones running on top of an RDBMS. Following on with the naming conventions of Neo4j, and its link to the 1999 film "The Matrix", I have called my completed tool *Reagan*[1].

Chapter 2 will be investigating the software engineering practices and reasoning behind my implementation choices, as well as a brief introduction to graph databases and some of the theory I plan to use.

Chapters 3 and 4 will expand further into the details of the tool, and how well the translated SQL queries perform when compared to Cypher. I will then conclude the dissertation with a summary of the work undertaken in Chapter 5.

[1]Reagan is the real name of the character Cypher, whose job was to free human minds trapped within the Matrix.

Chapter 2

Preparation

This section expands on the tools, theory, and software engineering practices that were used in the development of my project, setting the framework for conversion between graph and relational queries.

My starting point for this project was that I have never had any experience before with graph databases or the Cypher language. I did start with some knowledge of SQL and parsing, but both of these I had to learn in considerably more detail for the implementation, as I will outline shortly.

2.1 Graph databases

Graph databases such as Neo4j use the concept of nodes and relationships to model hierarchical data. Both nodes and relationships can have properties which can be queried. Relationships must have one direction and one type, but nodes can be labelled with zero or more labels. Figure 2.1 demonstrates visually some of these notions.

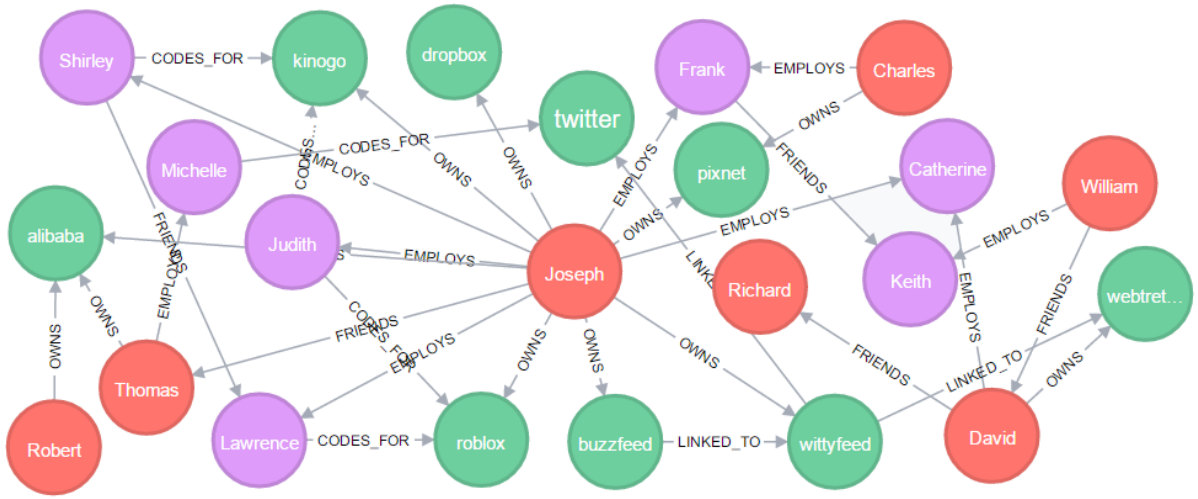

Figure 2.1: The basic structure of a graph database, using the pictorial conventions of the Neo4j browser, where circles are nodes, and lines are the relationships. The different colours refer to the different labels of the nodes.

Nodes can be thought of as the tuples we might see in a relational schema – they represent the *entities in the domain*. Relationships glue the individual items of data together, and thus represent a join in relational schemas (they could also be represented as a separate relation – i.e. the "OWNS" relation). Neo4j permanently stores these links, whereas they would either be computed at query-time on a relational database, or pre-computed using views.

Unlike relational representations, graph databases are schema-less. Whilst this has the benefit of reducing the barrier to entry for software engineers wanting to use these tools, neglecting database design can be costly. It will most likely lead to duplicate nodes being inserted unintentionally, or queries not returning the desired result. Neo4j (specifically Cypher) has commands that add constraints to the schema, such as enforcing the uniqueness property on some nodes.

Graph database management systems provide Create, Read, Update, and Delete (CRUD) methods on top of the graph model. Neo4j uses both *native graph processing* and *native graph storage* in its technology. Native graph processing benefits from traversal performance, but at the expense of making some queries that do not use traversals more computationally expensive [9].

Native graph processing also exhibits a property known as *index-free adjacency*. Each node maintains direct references to its adjacent nodes. Each node acts as an index to other nearby nodes, which is cheaper than using global indices. Query times, therefore, become independent of the graph size and are just proportional to the amount of graph searched. Neo4j implements this using doubly linked lists between nodes [9].

2.1.1 Graph query languages

Querying graph data requires a graph query language. There is, however, no equivalent SQL-like language that is portable between different graph database systems. Neo4j uses Cypher, a declarative language built by the same developers as Neo4j, and shares many similar features with SQL. Its constructs and simplicity arise from its closeness to English prose and its ASCII art representation for nodes and relationships. Cypher uses parentheses to denote individual nodes, and hyphens with an optional direction (indicated by either $<$ or $>$) to show the relationship.

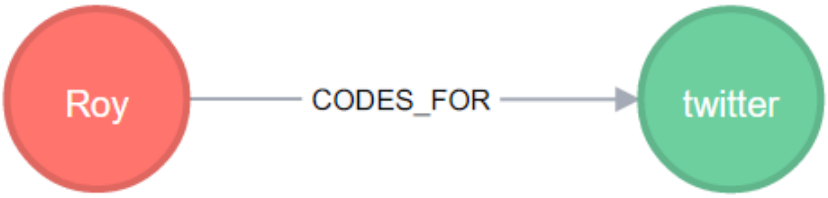

Figure 2.2: An example of how the Cypher input in Listing 2.1 is displayed visually on the Neo4j browser. In this example, Neo4j will return 'Roy' as one of the programmers who codes for Twitter.

Listing 2.1: Cypher query for Figure 2.2.

```
1 MATCH (a:Programmer)-[:CODES_FOR]->(b:Website {host:"twitter"})
2 RETURN a;
```

Recently, there has been a push towards a graph query specification (similar to how SQL is a standard of both ANSI and ISO). The *openCypher* project is aiming to fully specify Cypher, allowing it to be used as the query language for graph processing capabilities within any application[1]. Importantly, they provide the only available grammar file for Cypher, which I intend to use when parsing queries[2].

Neo4j also has the ability to execute Gremlin, a graph traversal language developed by the Apache Software Foundation[3]. Gremlin allows the programmer more control over the traversal of the query, and is written in a more functional style compared to Cypher. A major difference in Gremlin though is that it is executed on a graph virtual machine. This allows it to be system agnostic between graph vendors (such as Neo4j, Titan, OrientDB, etc.) [10].

This means that even beyond the Cypher to SQL conversion undertaken in this project, there is considerable scope for being able to "translate" between various graph engines/languages. Graph systems such as Cayley[4] do not even have their own storage engine, but rely on a separate backend store entirely. The work being conducted in this dissertation is, however, more focused on looking into what is required for such translations to take place – this also allows me to easily compare and contrast between different query types, representations, and graph structures.

2.1.2 Graph density

One metric that can be used to classify different types of graphs is graph density. The density of a graph measures how many edges (E) there are in total in the graph compared to the maximum possible number of edges between all the vertices (V). For a directed

[1] http://www.opencypher.org/
[2] https://s3.amazonaws.com/artifacts.opencypher.org/M04/Cypher.g4
[3] http://tinkerpop.apache.org/
[4] https://github.com/cayleygraph/cayley

graph, this is calculated by the formula:

$$D = \frac{|E|}{|V|(|V|-1)} \tag{2.1}$$

D can range in value between 0 and 1, with values near 1 approaching maximal density for a graph. These highly connected graphs may be modelled more effectively with a relational schema, as the overheads in Neo4j may become too high to be considered scalable, especially in light of a higher difficulty in partitioning graphs compared to RDBMSes. On the other hand, it may depend on what the value of V is as well.

2.1.3 Example graph

Figure 2.1 showed a small subset of the artificially generated graph for this dissertation. It was manufactured to examine all aspects of nodes and relationships, including nodes with multiple labels, relationships with properties, and cycles.

Red nodes in the figure are the 'owners' of websites (green). Programmers (purple) can code for websites, but can also be friends with other programmers and owners. An owner may also be a programmer. Websites are linked to one another too. Finally, owners can employ programmers to code.

The query in Listing 2.2 looks at the basic foundations of Cypher. It queries the following: "return a list of programmers, the website they are coding for, and the number of commits they have done for that website, where the owner of the website has more than 4 cars, and the programmer has amassed more than 93 commits."

Listing 2.2: Example Cypher query 1.

```
1 MATCH (coder:Programmer)-[job:CODES_FOR]->(site:Website)<-[:OWNS]-(o)
2 WHERE o.cars > 4
3 AND job.commits > 93
4 RETURN coder.fullname AS Hacker, site.name AS Webpage, job.commits AS
      NumberOfCommits;
```

Executing this query on a graph with 500 nodes and approximately 6,200 relationships returns the results in Table 2.1.

Hacker	Webpage	NumberOfCommits
Christina	popcash	97
Evelyn	google	94
Bruce	apple	94
Stephen	amazon	98
Raymond	cctv	96

Table 2.1: Results of Query 1.

Cypher can also describe nontrivial graph queries succinctly, and therefore queries where the solution in an equivalent relational schema is not immediately obvious. The

query in Listing 2.3 is expressing the following: "return the websites with the domain '.net', that the website 'google.co.uk' can reach with directed paths of length between 2 and 3". Executing on the same graph as before returns the results in Table 2.2.

Listing 2.3: Example Cypher query 2.

```
1 MATCH (a:Website)-[*2..3]->(b:Website)
2 WHERE a.name = "google" AND a.domain = "co.uk" AND b.domain = "net"
3 RETURN DISTINCT b.webID, b.name ORDER BY b.name ASC LIMIT 10;
```

b.webID	b.name
496	battle
253	csdn
289	daum
357	doubleclick
297	ettoday
346	fbcdn
460	gmx
231	pixnet
303	popcash
466	rolloid

Table 2.2: Results of Query 2.

2.2 Relational database technology

The relational model arranges data in terms of tuples, which are grouped into relations. Each tuple consists of an ordered set of attributes, where attributes themselves are an ordered pair of attribute name and type name.

First proposed by Edgar F. Codd, they are also characterised by their use of normalisation as a mechanism to eliminate any redundancy in the data, thus protecting the integrity of the data, and reducing any original relational schemas to "normal form" [11]. Relations in normal form must be joined together to return the correct information pertaining to a query. For this dissertation, I use the SQL `INNER JOIN` semantics for combining relations together.

Unlike how Cypher is specific to Neo4j, virtually all relational database technologies are queried using SQL. This standardised language is most likely one of the reasons why relational databases remain so popular to date.

2.3 Theory

2.3.1 Transitive closure

The transitive closure of a binary relation R is a relation R^+ defined as follows: xR^+y if and only if there exists a sequence:

$$x_0 = x, x_1, x_2, ..., x_k = y$$

such that $k > 0$ and $x_0\ R\ x_1$, $x_1\ R\ x_2$, ..., $x_{k-1}\ R\ x_k$ [12].

For example, if X is a set of websites, and xRy means "there is a direct path of links between website x and website y (where x and y are in X)", then the transitive closure provides you the set of all websites you can reach from any starting point.

Graph databases are optimised to a high degree for the transitive closure and related path-orientated queries [13]. Relational databases, on the other hand, cannot handle this directly. It is not possible to compute the transitive closure in the relational algebra, and can only be done in SQL using recursion, which is only supported by selected database engines. Transitive closures are important for the translation to SQL as they provide a mechanism for evaluating variable-length paths in a graph.

2.3.2 Hierarchical SQL queries

Relational databases and SQL are just fundamentally not expressive enough for recursive queries such as the transitive closure. But, a number of database engines provide facilities allowing SQL to handle hierarchical data. *SQL:1999* introduced new semantics that enabled recursion to be performed within CTEs (Common Table Expressions) [14].

A CTE is a temporary results set that exists for just one query. It can be used to simplify longer queries by breaking them down into smaller sub-queries. CTEs that have been previously defined in the same SQL statement can be referenced in subsequent ones. A final `SELECT` statement combines the CTEs with optional further predicates to produce the final results set [15]. With the addition of the optional `RECURSIVE` modifier, a CTE can be made to refer to its own output [16]. An illustration of the general framework for a CTE is included in Appendix A.

2.3.3 Joins in SQL

A fundamental SQL operator that will commonly appear in the translation of graph queries to a relational form is the inner join operator. The result of these joins can be defined as firstly computing the Cartesian product of all the tuples in both of the joining relations, and then returning all the results which satisfy the explicit join predicate. This Cartesian product operation can be extremely expensive, and so modern SQL implementations such as Postgres will typically use other approaches like hash joins.

2.3.4 Representing graphs in relational form

The most straightforward representation is arguably the most obvious one. All of the nodes are stored in one relation, *Nodes*, and all of the relationships are stored in a separate relation, *Edges*. This is demonstrated in Figure 2.3, and Tables 2.3 and 2.4.

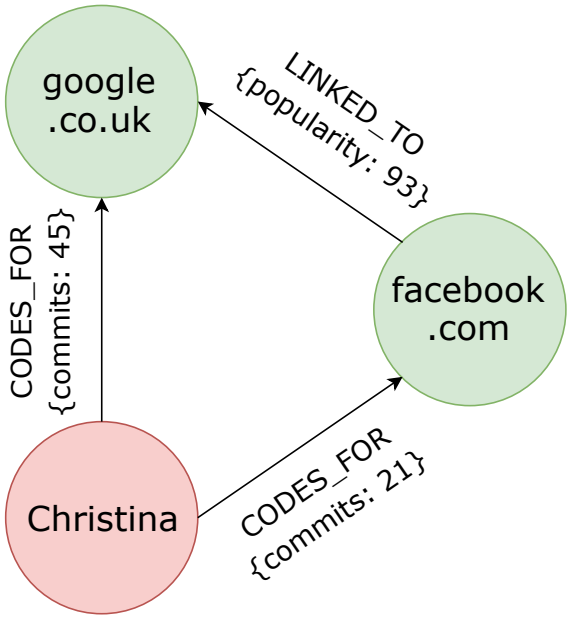

Figure 2.3: An example trivial graph.

id	label	name	host	domain
1	Programmer	Christina	*NULL*	*NULL*
2	Website	*NULL*	google	.co.uk
3	Website	*NULL*	facebook	.com

Table 2.3: Example *Nodes* relation for the graph in Figure 2.3.

left_id	right_id	relationship_type	commits	popularity
1	2	CODES_FOR	45	*NULL*
1	3	CODES_FOR	21	*NULL*
3	2	LINKED_TO	*NULL*	93

Table 2.4: Example *Edges* relation for the graph in Figure 2.3.

There are some optimisations and additional implementation aspects that I plan to use. Storing all of the attributes of the nodes in one relation, where there are multiple labels, will lead to a lot of `NULL` values populating the relation. To combat this, each individual label will also have its own relation. The translator tool decides at run time the optimal relation to join.

Furthermore, I plan to separate each relationship type into its own relation. Although this optimisation should be quicker, the trade-off will be roughly double the amount of disk space required to store all of the relations, as well as additional complexity in the translation tool.

There are then two different relational representations for graph functions I will investigate. *Option One* is based around the transitive closure theory outlined in §2.3.1, and *Option Two* is looking at modelling the queries with adjacency list relations.

Option One - Transitive closure

By storing permanently a relation that fully specifies the transitive closure of a graph, it is possible to use it in queries with joins in a relatively straightforward way. It also means expensive computation at query-time will be avoided. The drawback of this method is very large memory costs (even for smaller graphs) and increased difficulty in maintaining the transitive closure relation when updates to the graph occur [17].

Option Two - Adjacency list

Adapting the concept of *index-free adjacency* as mentioned in §2.1 can be achieved by creating an adjacency edge list for each node. In fact, two views will be created based on the existing *Edges* relation: one for edges leaving a node, and another one for edges directed to a node.

A potential benefit of this representation is that we reduce the number of expensive joins between relationships, and performance should, therefore, scale across larger graphs. One negative is that SQL is less suited for dealing with lists/arrays, compared to a native graph database engine.

2.4 Tools

2.4.1 Parsing Cypher

The tool requires a module that can accept Cypher queries as input, and parse them into an intermediate representation that can then be further translated into SQL later on.

Multiple options were available for this task. A hand-crafted tool would have been the most flexible, but reservations about the scalability of such a method for more complex queries meant this was not a viable option. Similarly, the existing parser for handling Cypher in Neo4j could have been adapted to fit the workflow of the tool, but I decided against this due to the high learning curve required to understand the Scala code that performs this.

Another option that was explored was to use *libcypher-parser*[5], a Cypher parsing library built in C. This library looked promising, but issues with both using it on Windows® and having to build an interconnect between this library in C and my tool in Java seemed too much work for the task involved.

[5] `https://cleishm.github.io/libcypher-parser/`

The adopted solution chosen was the ANTLR[6] (specifically the latest version, ANTLRv4) framework. ANTLR takes as input a grammar file that specifies a programming language, and from this, it can build a parser/lexer in another target language. The parsers can then be run with a Cypher input to automatically generate parse trees and abstract syntax trees, or tokenise the input for simpler parsing. It is therefore both flexible and scalable, and more capable of handling future language extensions.

2.4.2 Database engines

This project uses Neo4j Community Edition 3.1.1. The tool will communicate with Neo4j via the Java driver provided with the software. This will make the process of executing and timing queries easier.

The choice of relational engine was more flexible, but it required the power to perform recursive queries, be free and open source, and be able to handle large amounts of data. Postgres fitted these criteria and comes with good GUI tools and reliable JDBC[7] drivers for interfacing with Java. Scalability will also be very important, and Postgres 9.4 is known and used in the industry to serve large workloads.

2.5 Plan for implementation

2.5.1 Planned toolchain

Given what has been discussed so far, I have designed the tool in accordance to the following workflow demonstrated in Figure 2.4.

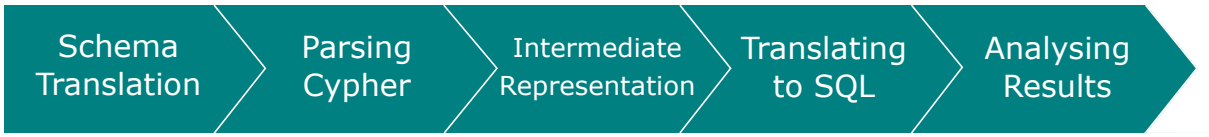

Figure 2.4: The five stages of the planned toolchain.

The *Schema Translation* module and the *Analysis of Results* are relatively independent modules in this chain, but nonetheless important. I will be writing the tool completely in Java, as the object orientated approach closely maps to how I intend to represent the Cypher query internally. Neo4j and Postgres are also easily compatible with Java, and this was a major factor in choosing the language. Further details of these individual modules will be outlined in Chapter 3.

2.5.2 Subset of queries to execute

The project could have the potential to convert all aspects of Cypher. This would lead to a more complete solution, but could become repetitive and lead to fewer interesting conclusions. Thus, it is important to choose a subset of Cypher to translate, and my plan

[6] Another Tool For Language Recognition
[7] Java Database Connectivity

for this is outlined below. The tool is based on the data from the latest Cypher reference card[8].

- All read only query structures, with the exception of the `OPTIONAL` keyword, which is where `NULL` values are used to fill in missing gaps in patterns. The `WITH` keyword will be translated, but only in limited scenarios and in certain predetermined patterns.
- The ability to create, update (using `FOREACH`), and delete nodes or relationships.
- Variable length patterns of the form `()-[*a..b]->()` and Neo4j graph functions such as `shortestPath((n1:Website)-[*1..6]->(n2:Website))`.

2.5.3 Software engineering methods

I have chosen a spiral-based model of development with evolutionary prototyping for this project. The tool itself I plan to build in four separate *sprints*, each one building on the code of the last sprint/prototype [18]. The release cycle for the prototypes will follow roughly what is outlined in Table 2.5.

Version 1 (*4 weeks*).	A prototype of the schema translator, and some basic translation modules implemented. No optimisations or additional features.
Version 2 (*6 weeks*).	A move to an automatic tool with fully finished schema translator. A larger subset of Cypher should now be able to be translated. Testing on smaller graphs to be completed.
Version 3 (*6 weeks*).	Improved performance and refactoring of existing code. Begin implementing either extensions and/or optimisations, or different representations of the translation model.
Version 4 (*3 weeks*).	Prepare the tool for testing and finalise the implementation. Refactor the code (if necessary) to be extensible for future work after the project.

Table 2.5: Planned Software Release Cycle.

The code will be hosted on GitHub as a backup and as my version control system. I also plan to regularly backup all files to both my personal OneDrive account and the MCS machines within Cambridge. I am using IntelliJ IDEA 2016.3.4 as my IDE for this project.

[8] `http://neo4j.com/docs/cypher-refcard/current/`

2.5.4 Requirements analysis

As a result of the preparation work, the following requirements were specified for the tool.

Requirement 1: The tool must be able to convert a wide range of Neo4j graph schemas to an equivalent relational database schema.

A 'dump' from the Neo4j console should be used as input for the schema conversion module, with the resulting SQL statement(s) being executed on the relational database engine. Additional metadata files may also be generated.

Requirement 2: The tool must be able to convert a modest subset of Cypher input to executable SQL automatically.

A text file or UI for allowing the user to enter Cypher queries, and generate the equivalent SQL. This assumes that the graph has already been converted as specified in *Requirement 1*.

Requirement 3: The tool must be able to show the execution times of running Cypher and SQL on Neo4j and Postgres respectively.

For thorough evaluation and testing purposes, multiple runs will be timed, with the average and standard deviation taken into account. Before the timed runs, there will be three dry runs to warm up the database caches.

Requirement 4: The tool must have a module that collates the results from both Neo4j and Postgres into text files, which can be compared for accuracy and completeness.

An important use case of the tool would be for Postgres to produce the same results as Neo4j.

Requirement 5: The tool should be coded to allow for extensions to the language to be included with minimal change to the whole code.

Given that it is unlikely the whole of the Cypher language will be converted for the project, a framework should be in place where further aspects of the language could be included without major refactoring.

Chapter 3

Implementation

This chapter will go into more detail of the success and challenges during the implementation phase, with diagrams and code snippets included where necessary. The outline of the sections approximately follows that of the toolchain in §2.5.1. There are then two additional sections looking at how I created artificial graphs for testing, and an explanation of the extension implemented.

3.1 Schema conversion

The first module in the toolchain is the one that converts 'dumps' from Neo4j into a relational schema. Initially, this was done in serial fashion, but as the performance degraded quite significantly with larger dump files, this module had to be converted to be more parallel than initially planned. The dump file for my largest artificial graph was 57MB, and for the largest OPUS graph I was testing, this became even larger at 91MB.

Dump files were obtained from the Neo4j shell by executing the `dump` command, and piping the results from the console output to a local text file. A dump will have the following form:

```
create (_994:‘Local’ {‘mono_time’:"6630", ‘name’:"2", ‘node_id’:994,
    ‘sys_time’:-460241671, ‘type’:4})
create (_995:‘Process’ {‘node_id’:995, ‘opus_lite’:true, ‘pid’:18849,
    ‘status’:1, ‘sys_time’:-460241671, ‘timestamp’:6662, ‘type’:3})

...

create (_0)-[:‘OTHER_META’ {‘state’:0}]->(_7)
create (_1)-[:‘PROC_OBJ’ {‘state’:10}]->(_0)
```

The dump required some manual editing before it could be parsed further, such as removing lines at the top of the file which is not relevant to the data. Additional parsing of the file is then carried out by the tool itself. This included removing unnecessary line breaks, as well as formatting characters for valid Postgres input – for example, ′ needed to be exchanged with ″.

The file is then read line by line into the tool, and distributed amongst 4 threads[1], each handling a quarter of the file concurrently. Each thread uses regular expressions to extract information about the nodes and relationships in the dump file.

When all of the threads have completed, the module joins all of the outputs back together into two files: *nodes.txt* and *edges.txt*.

These files are then used to create the *Nodes* and *Edges* relation in Postgres. Relations for each label type and relationship type present in the graph database are also created. Other important functions and materialised[2] views are also committed to the database at this point – an example of this module being run is shown in Figure 3.1.

[1]The value four was chosen as this was the number of available cores on the test machine – performance will be dependent on the amount of thread-level parallelism available to exploit.

[2]For execution on Postgres, the American spelling of 'materialized' is used.

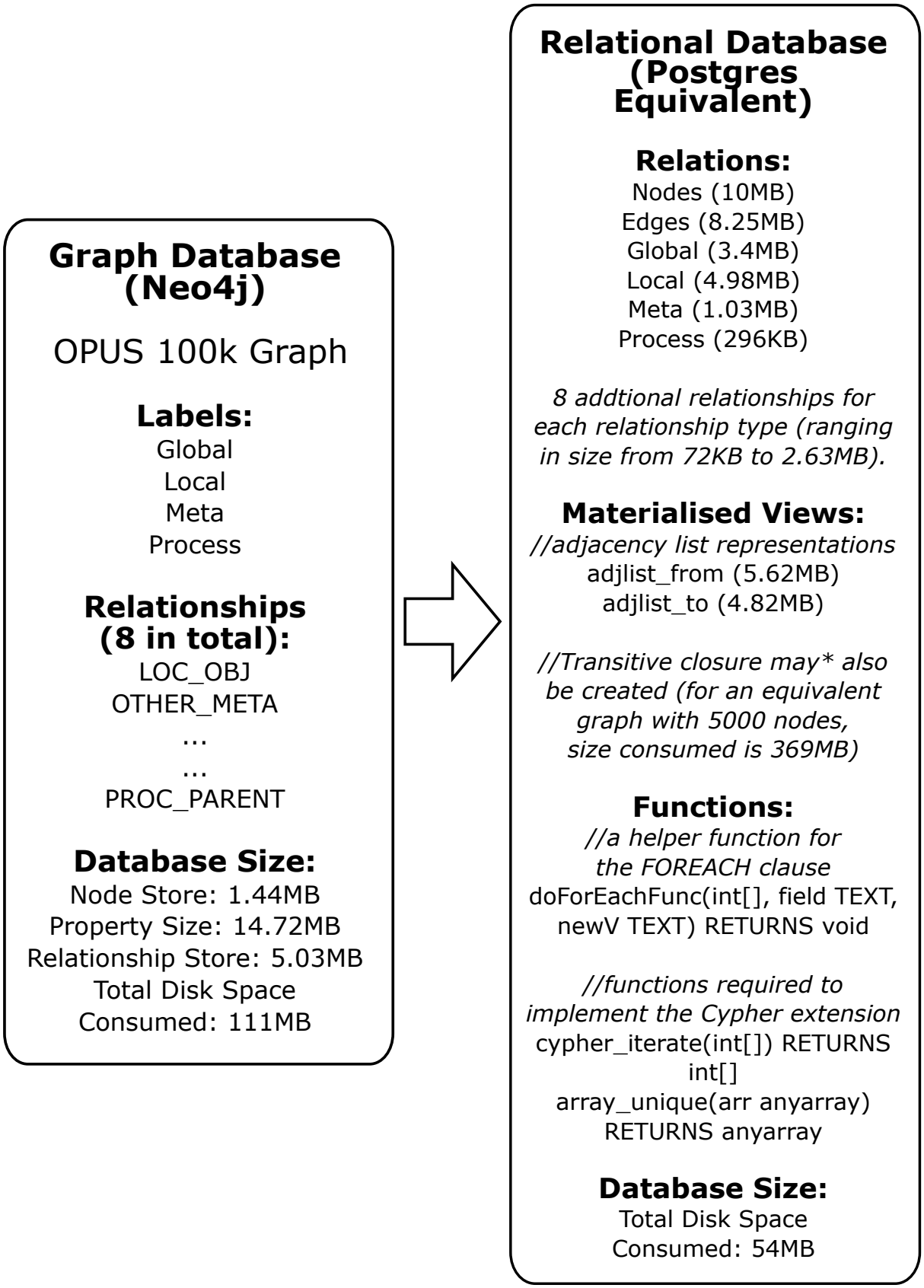

Figure 3.1: An overview of how a graph database is converted into a relational form, including the disk spaces consumed by both representations. *the transitive closure can only be built for smaller and sparser graphs.

3.2 Parsing Cypher to an intermediate representation

The input to the tool during translation is a standard text file with a list of Cypher queries to translate to SQL, and the name of the relational database to execute the SQL on. This assumes that the corresponding graph database is already being run as a separate process in Neo4j, and that the graph schema has been successfully converted. The output of the tool is what I refer to as a *DecodedQuery* object; it encompasses all the necessary information for the next step in the toolchain.

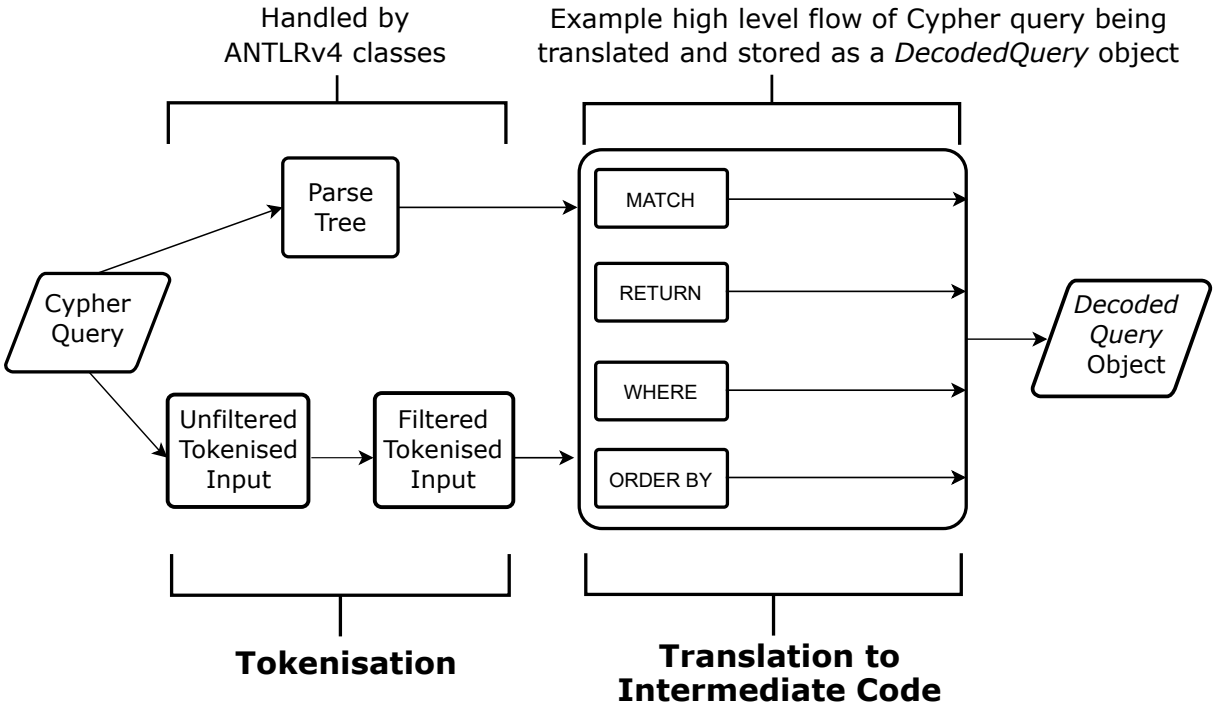

Figure 3.2: Closer inspection of the second part of the completed toolchain, detailing the role of ANTLR and how the Cypher query is parsed into a *DecodedQuery* object that I have specified.

3.2.1 Tokenisation

The initial phase of parsing Cypher uses the classes generated by the ANTLRv4 tool. A lexer for the Cypher language is first initialised and fed as input the Cypher query. The next step tokenizes the query into a list of tokens. Tokens are subsequently all reduced to lowercase, before finally removing all the unnecessary tokens such as `EOF` and `;`.

The query is also parsed using a parse tree to extract additional information from the input, such as the presence of certain keywords, and a mapping of any aliases in the `RETURN` clause. This is all stored in a *CypherWalker* object, which is used later on in the conversion to SQL. This object extends the *CypherBaseListener* class generated automatically by ANTLR.

Whilst ANTLR greatly simplifies the parsing process, I did find some issues and difficulties. Mainly, the grammar file itself needed some tweaking to work with the tool, including changing one of the words in the grammar, `return`, to `returnMain`, as the former is a protected keyword used by ANTLR. This did not affect how the Cypher input was entered, which was a positive.

3.2.2 Translation to intermediate code

Further use of the ANTLR framework was considered for better analysis of the query, but I opted instead to customise the next processing stage, in order to create a semantically rich intermediate representation designed specifically for the translation to SQL. It would be possible therefore for this module to be adapted accordingly for different RDBMSes if desired.

Intermediate representation

Consider Cypher queries that query the database (i.e. read-only). These at a minimum will have a structure of `MATCH ... RETURN ...`. The tool proceeds initially by attempting to split the token list into tokens that belong to the `MATCH` clause, and those that belong to the `RETURN` clause. Additional complications arise with other keywords, as will be outlined soon.

A `MATCH` clause consists of nodes and zero or more relationships. I extract these into internal objects that I have defined, which will store information such as its position in the clause, the ID it has, and any labelling or properties associated with them (including the direction of the relationship).

This proved taxing in some places. Extracting relationships from the query involved checking 19 different possible permutations (consisting of ID, type, direction, and any properties of the relationship). Likewise, a `RETURN` clause contains a comma separated list of the items to return from the query, which may include functions and the use of wildcards.

The complicated nature of some queries means that parts of the code have become difficult to extend. This is something the tool could definitely find improvement with. The most challenging aspect was parsing `WHERE` clauses. Making sure the boolean operator ordering was consistent, and accommodating for all the possible operators was tough, and further testing is required to ensure correct operation in all edge cases.

All the objects created in Java as a result of this module are bundled into a *DecodedQuery* object. The resulting SQL is also stored in this object for completeness.

Process of translation

The full process for parsing the Cypher query is below. It is only describing read-only queries; there is a similar process for queries that edit the database, but this will not be discussed:

- Work out the index within the token list for all the keywords.
- Extract from the token list the appropriate clauses for each section of the Cypher query.
- Decode the clauses into their relevant objects.
- Extract any additional information from the Cypher input, such as if `SKIP` or `LIMIT` are used, and if so storing the amount specified in both cases.
- Create a new *DecodedQuery* object.

3.3 Converting to SQL

The bulk of the project is dedicated to the generation of SQL. The intermediate representation discussed in §3.2.2 is the input to this module. I had initially planned this module in the toolchain as one class, but this became less manageable to fix as more rules and edge cases were added.

The alternative method I ended up implementing was to separate the individual types of queries out, into their own separate classes (discussed below). This also aided error fixing and correction.

All of the separate classes follow a similar pattern of creating the SQL statement, one logical chunk at a time. For example, when translating a query with multiple relationships, the CTEs are generated first, then the final `SELECT` clause is appended, finishing it off with any criteria in the `WHERE` clause.

3.3.1 SQL translation of Cypher

Single node queries

These are the most trivial to translate, and could in theory be computed by hand – nevertheless, my tool did add some additional features.

The relation being queried in the `FROM` part of the SQL query can be optimised. A simple implementation would just use the *Nodes* relation, but, as mentioned in §2.3.4, this is full of `NULL` values. If the user specifies a label in the Cypher query, then the query will use that relation (each label has its own relation in the relational schema). Even if the user does not provide a label, it is possible using the metadata gathered during the initial schema conversion to calculate the most optimal relation to use based on what the user is returning. For example, if the user wants all of the hostnames from a node, then the node must be of type *Website*.

```
MATCH (node:Website {domain:"co.uk"}) RETURN COLLECT(node.webID);
```

```
SELECT array_agg(n01.webid) FROM website n01 WHERE n01.domain = "co.uk";
```

In the above example, `COLLECT` translates in Postgres to `array_agg`. The length of the SQL query is not dramatically larger than its Cypher equivalent.

Queries with one or more relationships

When relationships are involved in the query, the SQL translation becomes less obvious. It will need to involve joins between relations, but additional constraints will also be needed to ensure consistency in the results.

Each individual relationship in the Cypher query is represented by a Common Table Expression (CTE), which is characterised by the `WITH` keyword in Postgres. These allow relationships to be built up iteratively by the tool using the same patterns – it also breaks up what would be a very complicated query down into its constituent parts [16].

```
//Full Cypher Query:
MATCH (a:Owner {state:"Texas"})-[:OWNS]->(b:Website)
RETURN DISTINCT b.host;

//Relationship:
(a:Owner {state:"Texas"})-[:OWNS]->(b:Website)
```

```
--Common Table Expression 'a':
WITH a AS (
   SELECT n1.id AS a1, n2.id AS a2, e1.* FROM nodes n1
   INNER JOIN e$owns e1 on n1.id = e1.idl
   INNER JOIN website n2 on e1.idr = n2.id
   WHERE (n1.state = "texas") AND n1.label LIKE "%owner%"
)

SELECT DISTINCT n01.host FROM website n01, a WHERE n01.id = a.a2;
```

The CTE is constructed using knowledge of the nodes in the relationship. This CTE is then referenced in the final line of the SQL query. The CTE is part of the same query that is executed at the same time – it is separated above for clarity.

As the tool accommodates for multiple labels, the tool needs to use the `LIKE` operator, as opposed to the simpler and cheaper equate operator.

To ensure the correct results are returned, knowledge of the `RETURN` clause in Cypher is needed. In the example above, the user wishes to return the hostnames of all the nodes with the ID `b` in the Cypher query. Thus, the SQL requires a condition which sets `n01.id = a.a2`. This restricts the output to just the nodes that should be returned.

Naturally, this idea extends to more relationships in the Cypher input:

```
MATCH (a:Programmer:Owner)-->(b:Website)<--(c:Owner)<--(a)
RETURN a.surname;
```

```
WITH a AS (
   SELECT n1.id AS a1, n2.id AS a2, e1.* FROM nodes n1
   INNER JOIN edges e1 on n1.id = e1.idl
   INNER JOIN website n2 on e1.idr = n2.id
   WHERE n1.label LIKE "%programmer%" AND n1.label LIKE "%owner%"
), b AS (
   SELECT n1.id AS b1, n2.id AS b2, e2.* FROM website n1
   INNER JOIN edges e2 on n1.id = e2.idr
   INNER JOIN nodes n2 on e2.idl = n2.id
   WHERE n2.label LIKE "%owner%"
), c AS (
   SELECT n1.id AS c1, n2.id AS c2, e3.* FROM nodes n1
   INNER JOIN edges e3 on n1.id = e3.idr
   INNER JOIN nodes n2 on e3.idl = n2.id
   WHERE n1.label LIKE "%owner%" AND n2.label LIKE "%programmer%"
   AND n2.label LIKE "%owner%"
)

SELECT n01.surname FROM nodes n01, a, b, c
WHERE a.a2 = b.b1 AND b.b2 = c.c1 AND n01.id = a.a1 AND a.a1 != b.b2 AND
    a.a2 != c.c2 AND a.a1 = c.c2;
```

The length of the SQL here is considerably more verbose than Cypher. When there is more than one relationship, the intermediate nodes must equate, hence the need for the conditions `a.a2 = b.b1` and `b.b2 = c.c1`. The node with ID `a` is being returned (`n01.id = a.a1`), and as this node appears in two separate relationships, once again they need to be made equal for the SQL to be logically equivalent to Cypher (`a.a1 = c.c2`).

The final complication to the SQL logic is to prevent cases where the results loop back on itself. In the above example, the conditions `a.a1 != b.b2` and `a.a2 != c.c2` are not strictly necessary, but the tool adds them anyway as I implemented the tool in such a way to be as general as possible for all queries.

Variable length path queries

In cases where the length of the path between two nodes can be variable, Cypher offers the following syntax:

Listing 3.1: Variable length Cypher query.

```
MATCH (a:Owner {state:"Wyoming"})-[*1..3]->(b:Website {domain:"com"})
RETURN COUNT(b);
```

The syntax `*1..3` allows for nodes between one and three hops from node `a` to be returned.

Two different implementations were attempted for handling these types of queries. My first attempt was to use the transitive closure. The transitive closure is built in SQL with the query in Listing 3.2, which builds the relation as a materialised view, allowing it to remain persistent between database connections.

Listing 3.2: SQL statements for the transitive closure of a graph.

```
CREATE MATERIALIZED VIEW tclosure AS (
   WITH RECURSIVE search_graph(idl, idr, depth, path, cycle) AS (
      SELECT e.idl, e.idr, 1, ARRAY[e.idl], false FROM edges e
      UNION ALL
      SELECT sg.idl, e.idr, sg.depth + 1, path || e.idl, e.idl =
          ANY(sg.path)
      FROM edges e, search_graph sg WHERE e.idl = sg.idr AND NOT cycle
   )
   SELECT * FROM search_graph WHERE (NOT cycle OR NOT idr = ANY(path))
);
```

The CTE uses the optional `RECURSIVE` keyword, which allows the CTE to refer to its own output, something very different to traditional SQL [16].

This worked perfectly well for small and simple graphs. Unfortunately, the generation of the transitive closure for graphs even of relatively small sizes (around 20MB) used over 500MB of space to build the transitive closure. This was not feasible.

I therefore implemented an alternative method using the adjacency list representation mentioned in §2.3.4. The adjacency list representation in Postgres makes extensive use of array functions in Postgres.

The adjacency list representation maps to what Neo4j internally does, and therefore it seemed sensible to attempt to emulate this behaviour. The doubly linked lists in Neo4j have been mapped as two similar but distinct relations in Postgres, as shown in Listing 3.3. One relation maps a node ID to a list of all the node IDs it is pointing to, whilst the other relation lists all of the nodes that point to a given ID.

The SQL translations of the Cypher query in Listing 3.1 using both models is included in Appendix B.

Listing 3.3: SQL statements for the adjacency list representations.

```
CREATE MATERIALIZED VIEW adjList_from AS (
   select idl as LeftNode, array_agg(idr ORDER BY idr asc) AS RightNode
   FROM edges e JOIN nodes n on e.idl = n.id GROUP BY idl
);

CREATE MATERIALIZED VIEW adjList_to AS (
   select idr as LeftNode, array_agg(idl ORDER BY idl asc) AS RightNode
   FROM edges e JOIN nodes n on e.idr = n.id GROUP BY idr
);
```

Shortest path queries

Functions in Cypher that are designed purely for graph structures required the most work to translate to SQL. The translation model for these types of queries again relies heavily on the adjacency list representation.

```
MATCH p=shortestPath((f {state:"California"})-[*1..3]->(t:Website))
RETURN t.host;
```

```
WITH a AS(
   SELECT unnest(rightnode) AS xx, 1 AS Depth, ARRAY[id] AS Path, id AS
       Start
   FROM adjList_from INNER JOIN nodes q ON leftnode = id
   WHERE q.state = "california"
), b AS (
   SELECT unnest(rightnode) AS xx, 2 AS Depth, a.Path || ARRAY[xx] AS
       Path, a.Path[1] AS Start
   FROM adjList_from INNER JOIN a ON leftnode = xx
), c AS (
   SELECT unnest(rightnode) AS xx, 3 AS Depth, b.Path || ARRAY[xx] AS
       Path, b.Path[1] AS Start
   FROM adjList_from INNER JOIN b ON leftnode = xx
), d AS (
   SELECT * FROM a UNION ALL SELECT * FROM b UNION ALL SELECT * FROM c
), finStep AS (
   SELECT n01.host, min(Depth), xx, Start FROM website n01
   INNER JOIN d ON xx = id WHERE label LIKE "%website%"
   GROUP BY host, xx, Start
) SELECT n01.host FROM finStep n01;
```

There is one CTE for each hop in the path. There are then two additional CTEs that need to be defined: one that aggregates all of the data from the previous individual CTEs, and then a final one to extract the correct fields for the query, and importantly for the shortest path query, to use only those nodes with the minimum depth value. Again, it is clear to see how the SQL query is very obscure compared to the original Cypher statement. The use of `UNION ALL` and extensive reliance on arrays in Postgres may not prove scalable with increasing path lengths.

Updating nodes using FOREACH

The FOREACH syntax in Cypher allows for data within a list to be updated. Once again, it provides very concise semantics where the immediate translation is not obvious.

```
MATCH (u:Programmer {state:"California"})
WITH COLLECT(u) AS ms
FOREACH (x in ms | set x.state = "CA");
```

```
SELECT doForEachFunc(array_agg(n01.id), "state", "ca")
FROM nodes n01
WHERE n01.state = "california" AND n01.label LIKE "%programmer%";
```

The translation mechanism for FOREACH relies on a predetermined SQL function that operates on arrays. The function is used whenever the FOREACH keyword is used. It takes three arguments: an integer array of IDs from all the nodes that are matched in the initial MATCH clause (in this example, all programmers from California), the field in the database to be updated (the state the programmer lives in), and the new value of this field. The full SQL of this function is included in Appendix C for the reader.

This example shows how using existing functions with some of the more advanced SQL features can not only keep the size of the SQL statement down, but also help to maintain some clarity in what the query is. The more challenging implementation aspects are abstracted away in the function.

3.4 Execution and timing

The timing of the queries will be an essential part of the evaluation. The time to initially create the graph will also need to be reviewed, but for the purposes of this project, I will assume this overhead will not be of concern.

The execution times of both Neo4j and Postgres were calculated from within the tool. I used Java's System.nanoTime() library method to record the total time.

For Neo4j, the results are returned lazily as a stream. The individual records can be accessed one at a time, but in order to more accurately match the Postgres JDBC driver, it is necessary to consume the whole results set, and time how long this takes.

```
// timing unit for Neo4j (notice the use of consume)
long startNano = System.nanoTime();
session.run(query).consume();
long endNano = System.nanoTime();
lastExecTime = endNano - startNano;

// timing unit for Postgres
long startNano = System.nanoTime();
ResultSet rs = stm.executeQuery(query);
long endNano = System.nanoTime();
lastExecTime = endNano - startNano;
```

For a more accurate set of times on both databases, a total of eight runs will be taken, with the average and standard deviation of the results being recorded for each separate query. To "warm up" the database (caching both databases for greatly improved performance), the first three runs are not recorded.

3.5 Reviewing the outputs

The first check performed by the tool is to count the number of records returned from both datasets. If this does not match, the translation must be incorrect, and an error is thrown by the tool. In the cases where the query is returning something other than tuples of data, for example, a `COUNT`, then 1 will always be the number of tuples returned, and hence this method of reviewing the outputs is pointless[3].

The second and more complete check is to compare the output of the individual records themselves. This requires a temporary store for both the outputs, and this is done in the form of simple text files. The text files, once filled with data, are compared using an Apache File Utility tool[4]. True is returned if and only if the files are an exact match.

An example output from the tool looks like the following:

```
**********
Cypher Input : match (a:Global)-[*1..4]->(b:Local) return b;
SQL Output: WITH a AS (SELECT ... WHERE n01.label LIKE "%local%";
Exact Result: false
Number of records from Neo4J: 26020
Number of results from PostG: 26020
Time on Neo4J:    321.978836 ms.
Time on Postgres: 214.739867 ms.
**********
```

This, however, encounters some issues, and occasionally returns false negatives. One of the main reasons for this was encoding issues between Neo4j and Postgres which I had not planned for. Moreover, my design choice of translating all of the data to lowercase letters early on in the toolchain and schema translation resulted in differences in sorting the tuples. If sorting is not used, results are returned naturally in different ways, and it is computationally infeasible to prove that two files necessarily produce the same outputs, but in a different order.

[3] In the case of `COUNT`, the files are checked to confirm that they contain the exact result returned from the function.

[4] `https://commons.apache.org/proper/commons-io/javadocs/api-2.5/org/apache/commons/io/FileUtils.html`

3.6 Creating artificial graphs for testing

During formative evaluation of the project, I arrived at a problem. The graphs I mainly used for initial testing were quite simple and small. The only thorough way to check all the aspects of the tool built would be to not only use the larger graphs provided by OPUS, but to test on some artificial graphs created from another program.

The tool takes two parameters: the number of vertices, and the type of graph (sparse, regular, dense) to build. From this, it creates a graph that adds edges proportionally to the number of edges a node already has. The full algorithm in pseudo code is outlined on the following pages.

Algorithm 1: Graph Creator

input : Number of nodes numVertices and graph density parameter densityType.
output: Adjacency matrix representation of the graph.

```
/* GetGraphDensity(densityType) returns a real number.          */
edgesToAdd ← numVertices * GetGraphDensity(densityType);
numEdges ← 0;
/* Initialise all elements in both arrays to zero/false.        */
degreeOfVertices[]← new Short[numVertices];
adjMat[][]← new Boolean[numVertices][numVertices];
for i ← 0 to edgesToAdd do
    if numEdges < edgesToAdd then
        indexFrom ← 0;
        indexTo ← 0;
        while !addEdge(indexFrom, indexTo) do
            indexFrom ← generate new random integer [0, numVertices);
            indexTo ← generate new random integer [0, numVertices);
        end
    else
        randomEdgeNum ← generate new random integer [1, numEdges];
        indexFrom ← 1;
        while randomEdgeNum > 0 do
            randomEdgeNum ← randomEdgeNum − degreeOfVertices[indexFrom];
            indexFrom ← indexFrom + 1;
        end
        addedEdge ← false;
        while !addedEdge && degreeOfVertices[indexFrom] ≠ numVertices − 1 do
            indexTo ← generate new random integer [0, numVertices);
            if addEdge(indexFrom, indexTo) then
                addedEdge ← true;
            end
        end
    end
end
```

Algorithm 2: Add Edge

input : Two integers, from and to.
output: True or False: True iff graph added to adjacency matrix successfully.

```
1 if adjMat[from][to] || from == to then return false;
2 adjMat[from][to] ← true;
3 adjMat[to][from] ← true;
4 degreeOfVertices[to] ← degreeOfVertices[to] + 1;
5 degreeOfVertices[from] ← degreeOfVertices[from] + 1;
6 numEdges ← numEdges + 1;
7 return true
```

The values for the different densities attempted to emulate those from real-world graphs. A 2006 conference paper by Melançon specifically looks into how to approach the density of graphs, and also provides some real-world density values [19].

Melançon uses a different measure of graph density to the formula mentioned in §2.1.2. The real-world graphs mentioned are categorised by their linear edge density, which is calculated by the ratio:

$$D = \frac{|E|}{|V|} \tag{3.1}$$

Graph Type	Real-World Example	Density Value
Sparse	Word Association (nouns in dictionary)	7.61
Regular	A social network or database (IMDb)	13.49
Dense	Links between web pages	25.67

Table 3.1: Example Graph Density Values.

Many of the networks I was attempting to emulate with my artificial graph module exploit the fact that nodes with a high number of relationships already are more likely to connect to other nodes. This will lead to cliques in sections of the network. For example, in a graph containing 2,000 nodes and 50,732 relationships, the maximal number of relationships attached to one node was 367. Nodes at the other end of the spectrum have just 13 links.

The output from this module is multiple CSV files containing information about which IDs belong to which nodes, and how they are connected. I manually added context to the CSVs to make them more complete for testing purposes. The context used was described briefly in §2.1.3, and was obtained from freely open source data. Website data about the most visited website was collected from Alexa[5]. Data for the cities and states was

[5] `http://s3.amazonaws.com/alexa-static/top-1m.csv.zip`

obtained from a large dataset about all cities in the USA[6], and some test names were gathered from a 1990 US Census[7].

Creating these graphs was computationally expensive. For a dense graph of 40,000 nodes, and therefore around one million relationships, loading this data from the CSV files to Neo4j took around 24 hours to create.

3.7 Cypher extension

My major extension to the project was to implement additional functionality to the graph model. I have phrased my extension in this way to make it clear that the Cypher query I describe below cannot *currently be executed* on Neo4j. I have created a template of what such a query may look like, and defined the semantics of how it would operate.

The additional feature is looking at querying repeating patterns within a graph. A user defines a pattern they wish to search, and a schema for how to perform the iterative matching. The query will store all of the nodes "touched" by the query, returning them similarly to traditional Cypher queries.

To make the extension simple, it was important to define the syntax in a clear and distinct manner, as demonstrated in Listing 3.4.

Listing 3.4: Simple example demonstrating the extended Cypher syntax.

```
1 ITERATE MATCH (a {host:"google"})-->(b:Website)-->(c {domain:"net"})
2 LOOP c ON b COLLECT n
3 RETURN n.host;
```

The `ITERATE` keyword makes it both clear that the Cypher will attempt to iterate over a graph, and it is used as a flag for my tool, so that it can expect to parse the query correctly. The remaining syntax encompasses the iterate pattern: `LOOP (1) ON (2) COLLECT (3)`.

`(1)` is the node ID holding the results of intermediate queries. `(2)` marks the starting point in the query for the previous results to start from. `(3)` is an extra identifier that is separate from the main query, and is what is used in the `RETURN` clause.

To demonstrate what this new syntax can do, consider the graph in Figure 3.3.

[6] https://raw.githubusercontent.com/grammakov/USA-cities-and-states/master/us_cities_states_counties.csv

[7] http://www2.census.gov/topics/genealogy/1990surnames/dist.all.last

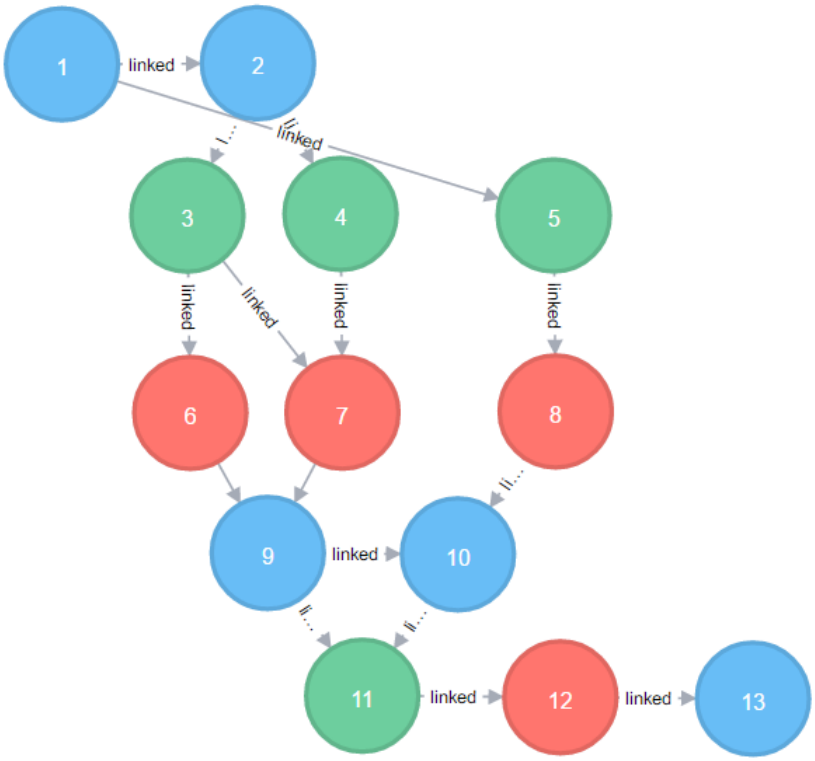

Figure 3.3: A trivial graph consisting of thirteen nodes with three different labels – blue for 'l1', green for 'l2', and red for 'l3'.

Figure 3.3 shows a common pattern, `BLUE --> GREEN --> RED --> BLUE`. Finding all the blue nodes at the end of this pattern *iteratively* would be hard to do with the existing Cypher syntax in a concise and clear manner. It is a more tractable problem with the `ITERATE` syntax:

```
1 ITERATE MATCH (a:l1)-->(b:l2)-->(c:l3)-->(d:l1)
2 LOOP d ON a COLLECT n
3 RETURN n.uid;
```

Executing this query on Postgres returns the results shown in Table 3.2.

n.uid
9
9
9
10
13
13
13
13

Table 3.2: Results of Iterate Query.

In comparison, running just the `MATCH` query on Neo4j once will return only six results. The extra two results that the `ITERATE` extension finds comes from the fact that the nodes with ID '9' and '10' are being used in another iteration of the `MATCH` clause. Hence the node with ID '13' is returned a further two times.

3.8 Summary of implementation

The tool built was designed with evaluation in mind. The individual modules were constantly being refined based on initial testing during the implementation, which allowed me to find any bugs or deficiencies in the code. The total codebase comprises 4,498 lines of Java (excluding 9,073 lines of Java generated by the ANTLR framework)[8].

Although I did discuss in my initial requirements analysis a possible GUI for the tool, it became more important to focus on providing a stable command line interface that was compatible with Unix instead, and this was successfully implemented.

Overall, this phase of the project went mostly to plan, with the only real issue being the difficulty in translating some aspects of Cypher. The *Schema Translator* required more work than I initially expected, but it was vital that this worked, as the rest of the toolchain relied on this being successfully implemented. A combination of the ANTLR framework and the intermediate representation I developed allowed for the bridge between Cypher and SQL to be made simpler. A testament to this was how I managed to seamlessly add increasingly more language features to the tool, as well as an extension to the Cypher language, without needing to greatly modify the intermediate representation. The majority of the implementation as a process flow is summarised in Figure 3.4 on the following page.

[8]Source line count calculated using the *Statistic* plugin for IntelliJ – (`https://plugins.jetbrains.com/plugin/4509-statistic`). The count does not include comments or blank lines.

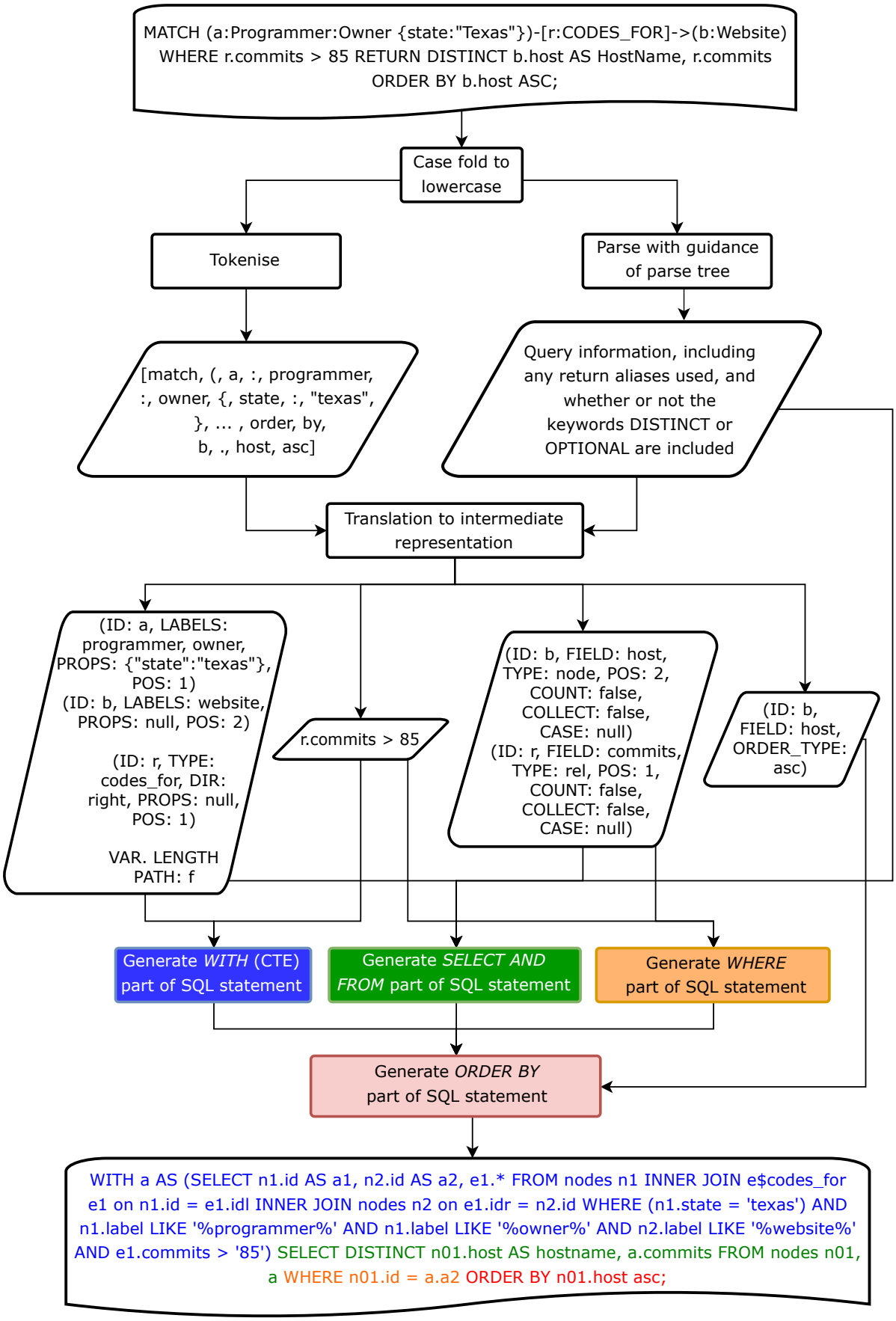

Figure 3.4: Process flow diagram of the translation unit, from the input of an example Cypher query, to the output of the equivalent SQL.

Chapter 4

Evaluation

This chapter will be discussing the results collected from the running of the tool on carefully constructed experiments, concluding with a critique of the project against the original success criteria.

4.1 Overview

The performance and scalability tests were run predominantly on a machine based within the Computer Lab. The specifications of the test machine were:

- 8× Intel® Xeon® E3-1230 V2 @ 3.30GHz (Quad-Core)
- 46GB free hard drive space
- 16GB RAM
- Ubuntu 15.04 Operating System
- Software:
  - OpenJDK (running Java 1.8)
  - Neo4j-Community Edition 3.1.1
  - Postgres 9.4.5

The evaluation was performed on a set of predefined queries, one graph at a time. They are all initially executed three times without being measured, to prepare the caches of both databases and eliminate any erroneous performance quantities from being collected (the execution times may be considerably higher than the *normal* runs). Each query is then timed for five individual runs. The results are stored in the Postgres database for each graph.

For ease of testing, I executed a shell script on the machine which automatically ran the tool against all of the databases sequentially, sending the results back via email for analysis.

4.1.1 Experiment plan

There were four main experiments that I ran in order to obtain a general overview of both databases. These are discussed in more detail in the following sections, but I outline them below:

- Evaluating the relational data representations.
- Performance and scalability of Cypher graph queries on the databases.
- Investigating graph based functions and algorithms.
- Evaluating more complex/typical graph queries.

Graph ID	# of Nodes	# of Relationships	Neo4j Size[1]	Postgres Size
Rand_10k_Sparse	10,000	75,372	25MB	18MB
Rand_10k_Dense	10,000	253,503	123MB	37MB
Rand_40k_Sparse	40,000	301,490	166MB	49MB
Rand_40k_Dense	40,000	1,014,086	513MB	125MB
OPUS_5k	5,000	7,655	6.2MB	379MB[2]
OPUS_20k	20,000	29,498	22MB	17MB
OPUS_100k	100,000	154,866	111MB	54MB
OPUS_400k	400,000	812,028	488MB	219MB

Table 4.1: Graphs under evaluation.

4.1.2 Evaluation of schema translator

Looking at the *OPUS_100k* graph specifically, extracting a dump of this graph produces a 19.81MB text file (the size of the Neo4j graph database on disk is 111MB).

The initial preprocessing of the file to remove line breaks and appropriately format the characters took 188.39ms. The processing of the file into its nodes and relationships using eight threads took 1,680.95ms.

The bulk of the time is in committing the relations to Postgres, outlined in Table 4.2. Overall though, this initial overhead is not considerably large to be of concern. Moreover, the total database size in Postgres turns out to be 54MB, which is not only acceptable, but less than half the size of the corresponding Neo4j database.

[1]Neo4j graph database size calculated by measuring the disk space used to represent it: `du -sh opus_400k.graphdb/`

[2]Size of the database inflated due to the size of the transitive closure, which is 369MB alone.

Description of SQL	Execution Time (ms)	Notes
Create *Nodes* relation.	70,088.89	The time to create the *Nodes* SQL statement takes an additional 2,101.96ms.
Create *Edges* relation.	97,032.60	The time to create the *Edges* SQL statement takes an additional 434.62ms.
Create separate relations for each node label type.	140.78	-
Create separate relations for each relationship type.	283.71	-
Create adjacency list representations.	1,316.26	This includes committing two separate materialised views: one for relationships pointing towards each node, and one for all the relationships originating from each node.
Create any additional relations.	50.08	This includes three functions and one sequence to help both the Cypher `FOREACH` keyword, and to assist with the extension that uses the `ITERATE` keyword.
Total execution time.	169,196.03	-

Table 4.2: Time to execute relational schema on Postgres for *OPUS_100k*.

4.2 Results

4.2.1 Experiment One: Evaluating the relational data representations

The two ways implemented for modelling the graph data included having the transitive closure as a relation in Postgres, and also modelling the relationships as two separate adjacency lists. This test was looking at how they both perform when executed on the same set of Cypher inputs.

As it transpired, the transitive closure could only be feasibly built for the sparser graphs I had. I performed this experiment on the *OPUS_5k* graph only. Computing the transitive closure even for a modest number of nodes or relatively high density will take an unreasonable amount of time, and can not be considered a feasible solution.

The Cypher query being evaluated was a variable path-length query between two nodes, both with labels. Path lengths between 2 and 50 were measured. The results are shown in Figure 4.1.

Listing 4.1: Cypher query for experiment one.

```
1 // n varies between 2 and 50 for this experiment.
2 MATCH (a:Global)-[*1..n]->(b:Local) RETURN COUNT(b);
```

The initial cost of the query being executed on the transitive closure representation is much higher than the other representations. There is some fixed cost in using the large relation, which is being reflected in the queries with a maximal path length of less than 10.

However, as the path length increases, and the number of nodes being returned increases, the rate of increase in execution time is smaller for the transitive closure than on both Neo4j and the adjacency list representation. At a path length of around 12, Neo4j becomes the slowest. Neo4j continues this upwards trend as the path length increases. At a path length of around 30, it is almost twice as slow as both the relational representations.

More fascinating though is that at path lengths greater than 30 on this query, the transitive closure representation becomes the most optimal form. Whilst this may be the case, it is unlikely in practice of such a case where a user would wish to write a query like `MATCH (a:Global)-[*1..40]->(b:Local) RETURN COUNT(b);`.

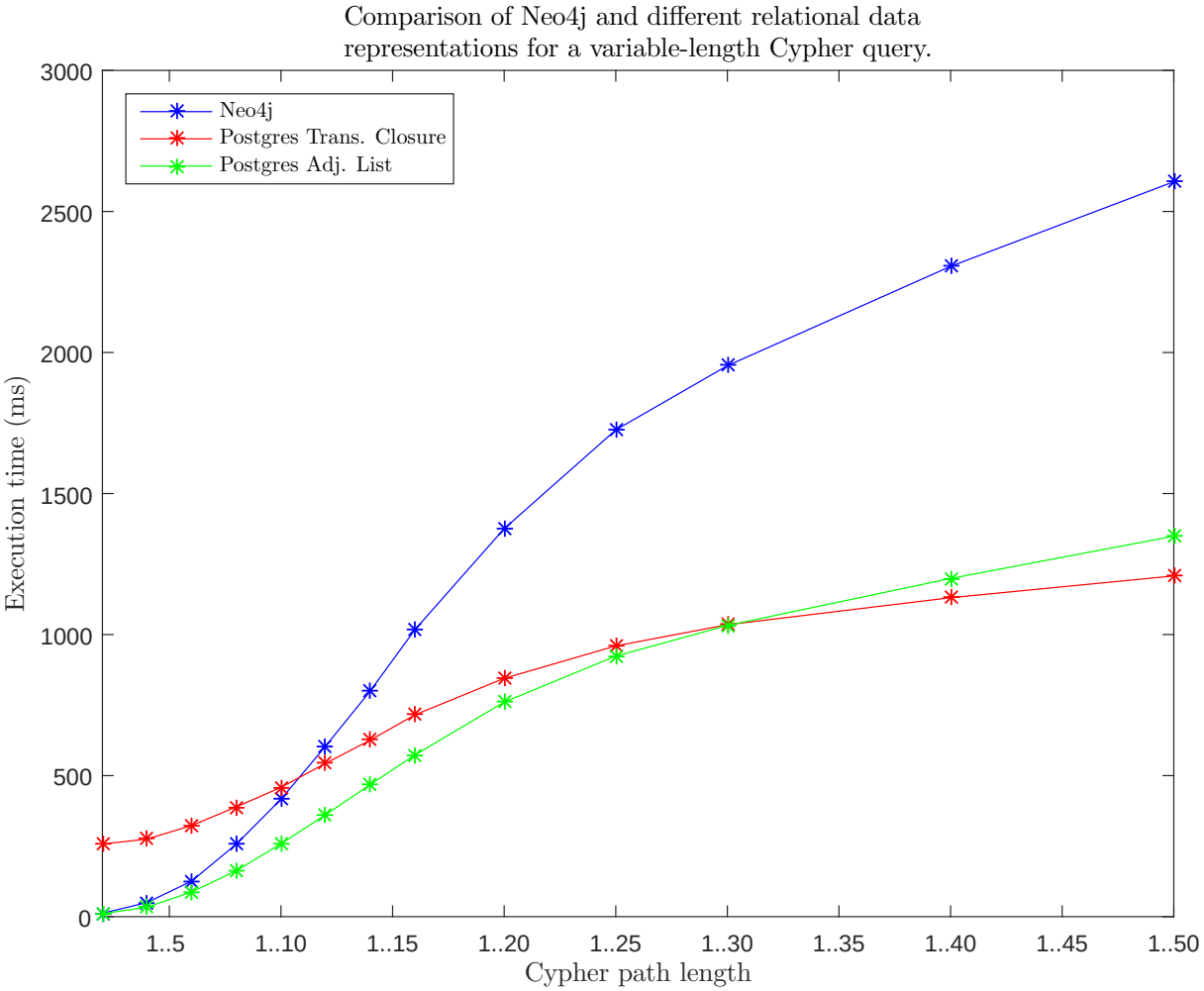

Figure 4.1: Results of experiment one.

Experiment 1a: Testing on larger graphs

Although it was infeasible to compute the whole transitive closure, it was possible to model a *partial transitive closure* on Postgres by limiting the depth of the recursive function. The results for the *OPUS_100k* and *OPUS_400k* graphs are shown in Figure 4.2.

We now see that initially (up to depth 3 on the path length), Neo4j performs best. The overhead of the transitive closure is still too great – add to this the fact that the transitive closure even up to a depth of 5 on the *OPUS_400k* graph takes up 1.9GB of disk space, this solution is not practical.

As a result of these experiments, I will now only focus on using the adjacency list representation developed. This representation looks the most scalable and appears a lot better from a performance aspect.

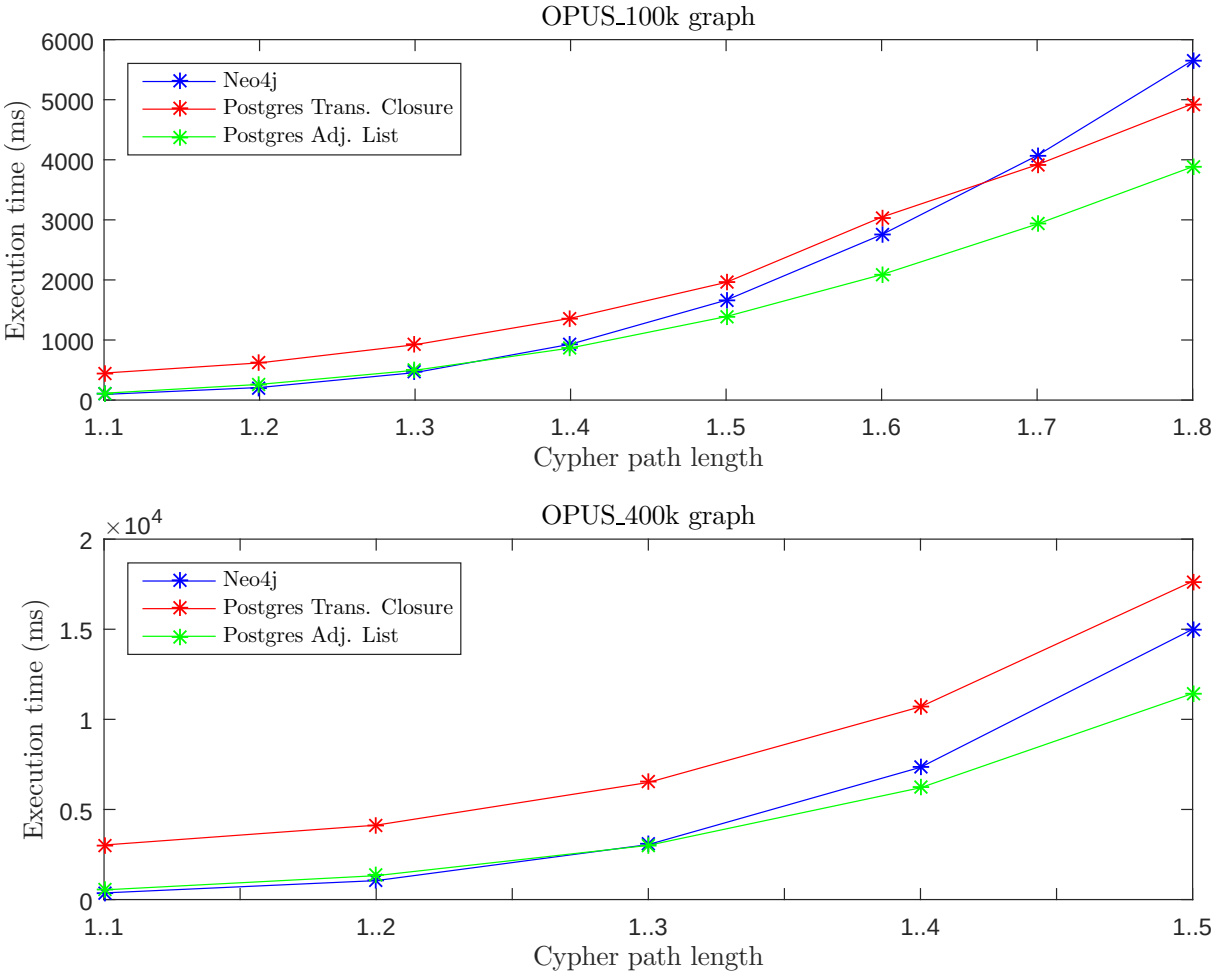

Figure 4.2: Results of experiment 1a. Note the different scales on the x-axis due to the limitations in creating the transitive closure on increasingly larger graphs. The query being executed is the same as the one in experiment one.

4.2.2 Experiment Two: Performance and scalability of Cypher graph queries on the databases

Cypher provides a very natural and familiar syntax for querying relationships. A programmer would *mostly* want to write concise and sensible Cypher, as opposed to a complicated, join based SQL statement.

The performance of the queries though is another matter. To evaluate this, I used the query in Listing 4.2, testing on all available graphs (except *OPUS_400k*).

Listing 4.2: Cypher query for experiment two.

```
1 // Notice how the query loops back on itself (ID 'a').
2 MATCH (a)-->(b)-->(c)<--(a) RETURN COUNT(a);
```

The data collected and shown in Figure 4.3 highlight some interesting results. On the OPUS graphs, Postgres performed much better than Neo4j. For the *OPUS_100k* graph, Postgres performed 20× better.

However, when the number of relationships is considerably greater than the number of nodes, the performance of Neo4j was a lot better. Even for the smallest sparse graph of 10,000 nodes, Neo4j was faster, and it outperformed Postgres massively on the denser graphs.

An explanation for this could be that the `JOIN` operation being used in SQL became too expensive when the number of relationships gets too large. Looking at the *Rand_10k_Sparse* and *Rand_10k_Dense* specifically – they both contain the same number of nodes, but with the denser graph containing ∼4× more relationships. The increase in execution time on Neo4j between these two graphs was 18× greater, whilst on Postgres, it was 27× greater.

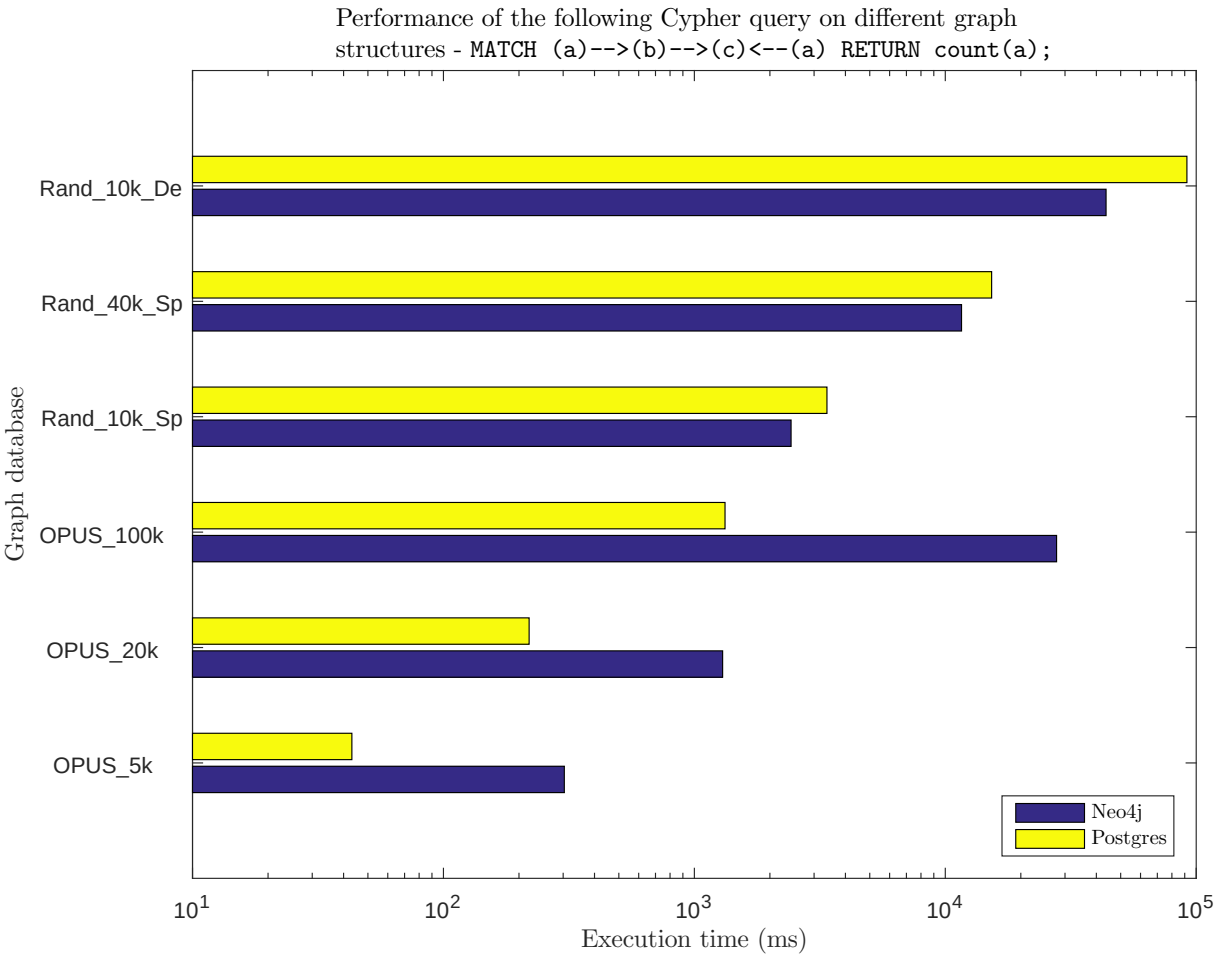

Figure 4.3: Results of experiment two. Execution time on x-axis in base-10 logarithmic scale.

Using labels as indexes

The effect of adding labels to a query in Neo4j is very dramatic. On the OPUS graphs tested, performance drastically swings from being better on Postgres to being better on Neo4j. A more in-depth look into why labelling improves the execution planner of Neo4j is discussed during the next experiment. The results of the query with the additional labels are shown in Figure 4.4.

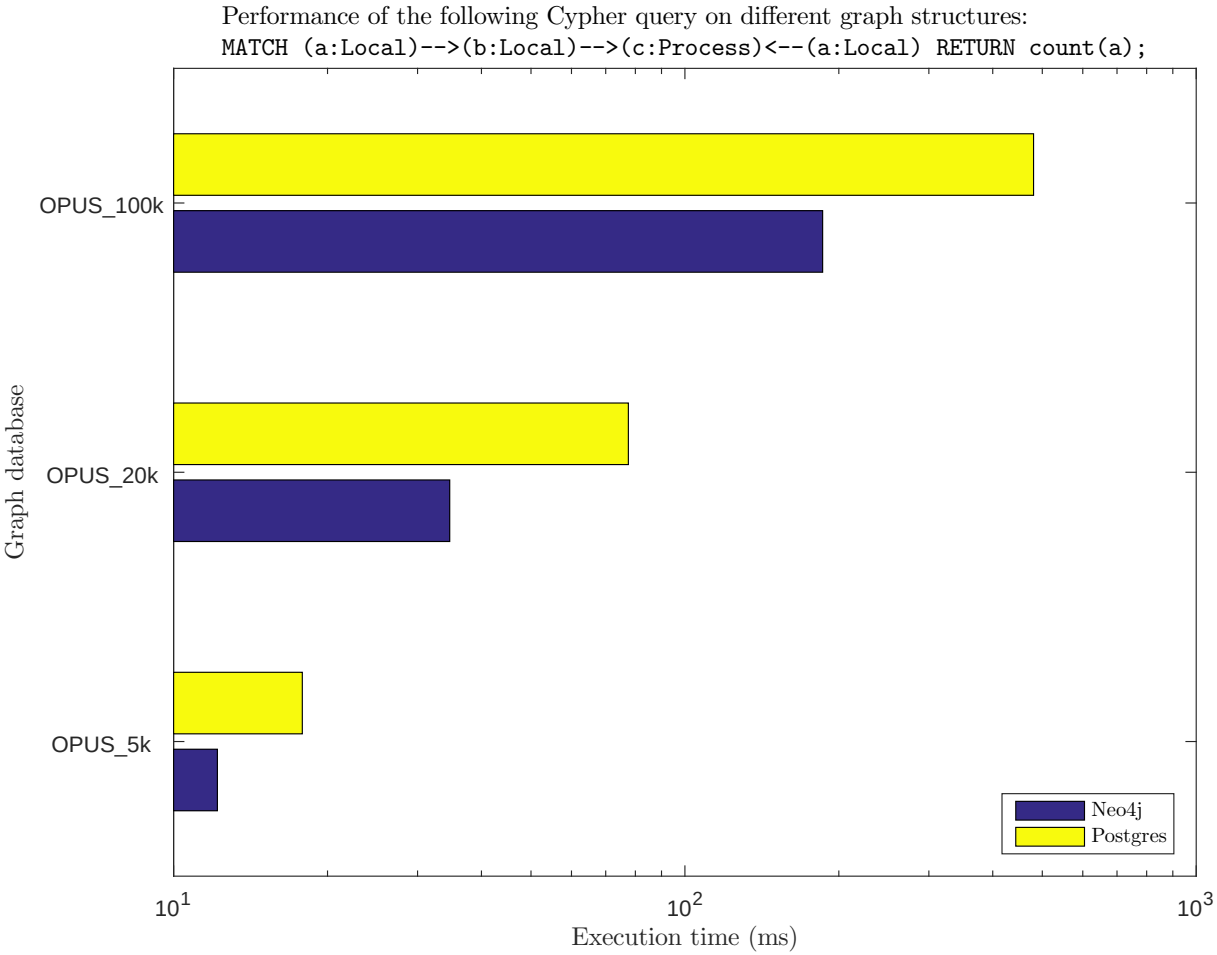

Figure 4.4: Results of running the query `MATCH (a:Local)-->(b:Local)-->(c:Process)<--(a:Local) RETURN count(a);` on selected OPUS graphs. Note the addition of labels in the query compared to the query being evaluated in Figure 4.3. Execution time on x-axis in base-10 logarithmic scale.

4.2.3 Experiment Three: Investigating graph based functions and algorithms

Finding the single shortest path between two nodes is not only useful in a lot of graph applications, but also easily computable in Cypher with the `shortestPath` function. Once again, a programmer would only want to use the Cypher syntax, as opposed to writing out the SQL equivalent.

The difference between running the queries on the OPUS graphs and the artificially created random graphs was so large, it is vital to study them separately.

OPUS graphs

OPUS graphs are very highly connected, and this provides one explanation as to why they suffer from performance degradation in certain scenarios.

Listing 4.3: Cypher query for the OPUS graphs for experiment three.

```
1 MATCH p=shortestPath((f {name:"omega"})-[*1..3]->(t))
2 RETURN COUNT(t.node_id);
```

The query is attempting to find all nodes of any type that are between one and three directed relationships away from any node with the name 'omega'. The *OPUS_5k* graph has 316 nodes with the name 'omega'; the *OPUS_20k* graph in comparison has 1,867. It became quickly apparent that the query being executed on Neo4j was not scalable past 20,000 nodes.

	Execution Time (ms)	
Graph ID	Neo4j	Postgres
OPUS_5k	7,285.46	14.63
OPUS_20k	18,1535.93	66.21

Table 4.3: Results of experiment three on selected OPUS graphs.

The results in Table 4.3 show an incredibly large difference between the performances of the databases. Even on relatively small databases, a simple to understand, and possibly very useful graph based query, is significantly better on the relational database. The performance of Postgres and its adjacency list representation is considerably higher than even optimistic predictions. On the *OPUS_20k* graph, Postgres was over 2,700× faster.

We have to question why there is such a large difference. The database execution plans of both Neo4j and Postgres will provide some insight. Thankfully, both of the systems have tools available to analyse these plans[3].

The Neo4j planner firstly scans all of the nodes, and then filters them on the name 'omega'. However, no filtering information is provided for the other node with the label `t`. Neo4j ends up performing the *Cartesian Product* of the 316 nodes with all of the nodes in the database that could be reached (4,957). This leads to a temporary database store

[3]Neo4j uses the keyword `PROFILE`; Postgres uses `EXPLAIN ANALYZE`.

with 1,566,412 results, which it then performs the shortest path calculation on. In total, there are over 3 million database accesses.

The plan from Postgres is harder to analyse. The text based output can be made simpler through graphical interfaces such as the GUI for Postgres, or through some online tools[4]. The main thing to notice though is that the join operations are very fast. Before each `INNER JOIN`, the relation being joined is hashed on some condition. Therefore, despite the fact there are four joins occurring, they are not the slowest part of the query.

The most expensive operations are the appending of the three CTEs built up during the query, and the sorting operation when choosing the correct *shortest paths*. The sort in Postgres uses an in-memory quicksort for maximal performance.

Using this additional knowledge, I reviewed how the databases change their execution plans when more knowledge is provided. I timed two more Cypher queries, each one containing successively more indexing information.

```
1 // Experiment 3a (addition of 'Local' label to node with id 't')
2 MATCH p=shortestPath((f {name:"omega"})-[*1..3]->(t:Local))
3 RETURN COUNT(t.node_id);
4
5 // Experiment 3b (addition of property 'ref_count' with value 2)
6 MATCH p=shortestPath((f {name:"omega"})-[*1..3]->(t:Local {ref_count:2}))
7 RETURN COUNT(t.node_id);
```

	Execution Time (ms) Exp. 3a		Execution Time (ms) Exp. 3b	
Graph ID	Neo4j	Postgres	Neo4j	Postgres
OPUS_5k	4,463.40	10.13	691.75	9.69
OPUS_20k	76,783.80	45.72	12,504.86	41.63

Table 4.4: Results of experiment 3a and 3b.

The additional of both a label and a property reduces the execution time of the query by 93.1%, whereas Postgres can only reduce its execution time by 37.1%. The Neo4j execution plan has managed to halve the number of database accesses with this new query, and more importantly, the expensive *Cartesian Product* operation now only has a result of 3,160 rows, as opposed to 1,566,412 before. The Postgres execution plan remained identical, with the speed-up being achieved by having to sort and aggregate fewer tuples than before.

All of the execution plans referred to above are included in Appendix D.1 and D.2.

[4] `https://explain.depesz.com/s/kyyg`

Random graphs

A similar query was executed on the random graphs as before, changing the parameters where necessary. I found this query was computable over all the artificial graphs created.

Listing 4.4: Query for the artificial random graphs for experiment 3c.

```
MATCH p=shortestPath((f {state:"California"})-[*1..3]->(t:Website
    {host:"google"}))
RETURN COUNT(t.node_id);
```

	Execution Time (ms)	
Graph ID	Neo4j	Postgres
Rand_10k_Sparse	1,187.90	154.00
Rand_10k_Dense	2,181.80	6,034.37
Rand_40k_Sparse	12,833.85	634.78
Rand_40k_Dense	22,032.43	24,928.22

Table 4.5: Results of experiment 3c on random graphs.

Unlike previously when the tool could consistently outperform Neo4j on all graphs, there is some disparity in the performance based on the density of the graph. On both of the sparse graphs, Postgres performed considerably better, but was slower on the denser graphs. Saying this, however, the gap between the execution times narrowed dramatically between the graphs with 10,000 nodes and those with 40,000. If larger random graphs were available, I would expect Postgres to eventually outperform Neo4j on all graph types (at least up until Postgres uses all available memory and breaks).

4.2.4 Experiment Four: General evaluation on other read based queries

The Cypher query in Listing 4.5 represents a typical graph query that a user *may* write using the `WITH` keyword. `WITH` enables query parts to be chained together, piping the results from one to be used as starting points or criteria in the next [16]. This allows for filters on aggregated function results. Conveniently for the translation process, the use of `WITH` in SQL is already common within the translation process, in the form of CTEs. However, the tool built can currently only use the Cypher semantics of `WITH` in certain predetermined scenarios.

The query in Listing 4.5 is stating the following: "count all the links between a single 'global' node and all of the 'local' nodes it can attach to, with any relationship. Return only the node IDs of all the 'global' nodes where they have two or more relationships, ordering by the number of relationships in a descending order".

Listing 4.5: Cypher query for experiment four.

```
MATCH (a:Global)-[m]->(b:Local) WITH a, COUNT(m) AS Glo_Loc_Count
WHERE Glo_Loc_Count >= 2
RETURN a.node_id, Glo_Loc_Count ORDER BY Glo_Loc_Count DESC;
```

Listing 4.6: SQL equivalent statement for the Cypher query in Listing 4.5.

```
-- 'global' and 'local' are relation names and not keywords.
-- 'timestamp' and 'value' are properties and not keywords.
CREATE TEMP VIEW wA AS (
   WITH a AS (
      SELECT n1.id AS a1, n2.id AS a2, e1.* FROM global n1
      INNER JOIN edges e1 on n1.id = e1.idl
      INNER JOIN local n2 on e1.idr = n2.id
   )
   SELECT n01.*, count(a.*) AS glo_loc_count
   FROM nodes n01, a WHERE n01.id = a.a1
   GROUP BY n01.name, n01.opus_lite, n01.mono_time, n01.pid, n01.sys_time,
       n01.type, n01.status, n01.id, n01.timestamp, n01.node_id,
       n01.ref_count, n01.label, n01.value
);

SELECT node_id FROM wA WHERE glo_loc_count >= 2
ORDER BY glo_loc_count desc;
```

This query could be tested on all of the OPUS graphs, and the results can be seen in Figure 4.5. The figure also includes the execution times of just running the relationship in the original Cypher query – `MATCH (a:Global)-[m]->(b:Local) RETURN a.node_id`.

On this query, Neo4j performs better on all graphs tested for the whole `WITH` query. I even tested a similar query on a more complicated graph pattern structure with more links, to see if Postgres could perform any better, but the trend of the results remained

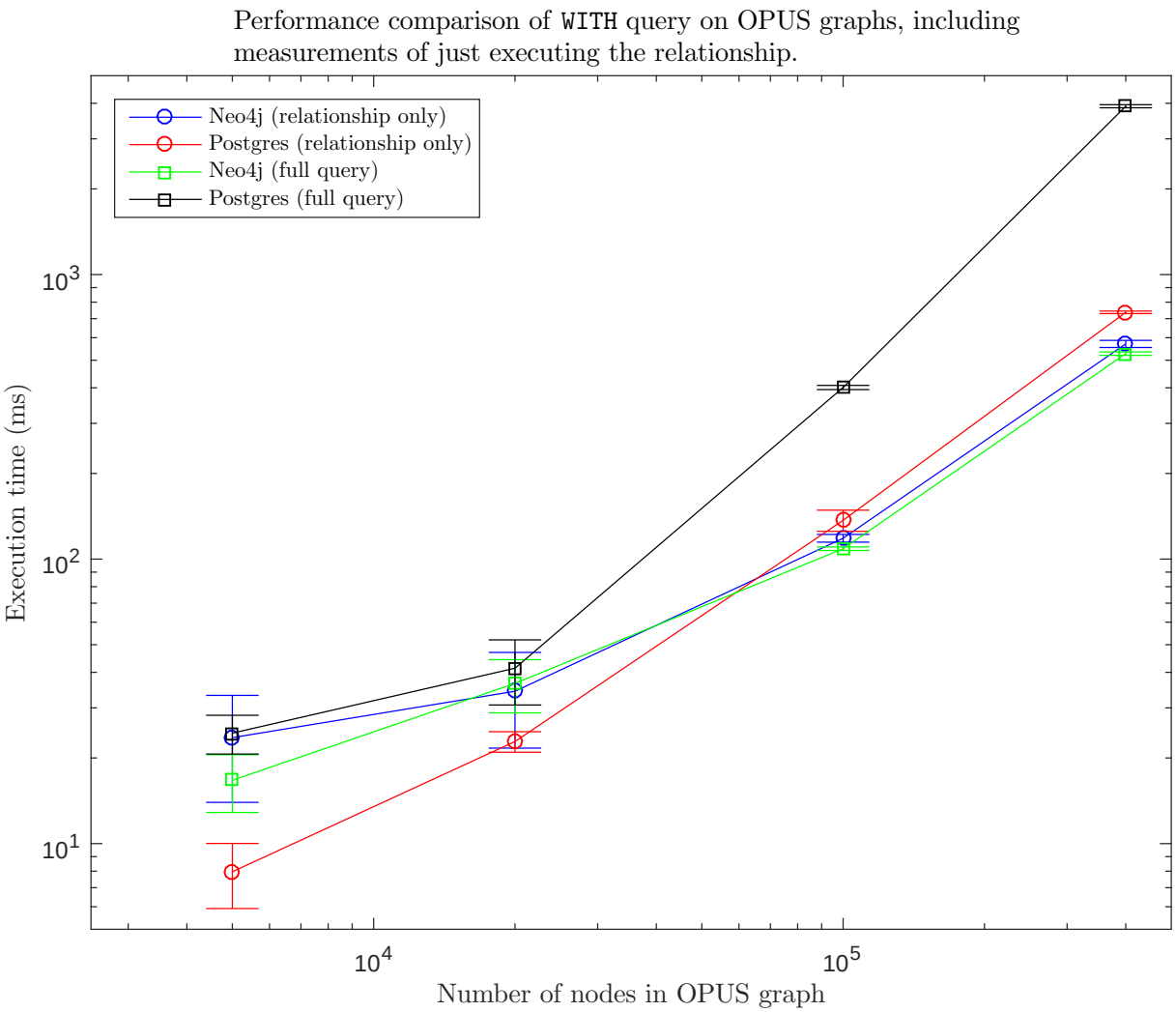

Figure 4.5: Results of experiment four. Both axes are in base-10 logarithmic scale. Error bars included to show the standard deviation of the mean of the five results collected. The circle plots are for the execution of *just* the relationship; the square plots are the execution times for the whole Cypher input.

the same. It is therefore possible to argue that whilst so far the relational schema has been seen as a very good tool for graph queries, it has scenarios where it cannot compete.

Perhaps more interestingly though is how Neo4j appears to perform better on a more advanced query. On the *OPUS_400k* graph, the query containing `WITH` was 43ms faster than just obtaining the results without any further processing. The relational representation is able to match the performance of Neo4j when just the relationship is executed – in fact, it is faster for both the *OPUS_5k* and *OPUS_20k* graphs.

The current translation mechanism for `WITH` firstly creates a temporary view in Postgres that stores all of the tuples returned from the relationship part of the query. For the `COUNT` to be completed, it is required in SQL to `GROUP BY` all of the attributes in this temporary view. This view is then referred to by a relatively straightforward and concise SQL query that satisfies the ordering constraint and `WHERE` predicate.

This *current approach* results in very expensive sorting and group aggregation operations within Postgres that cause the drastic increase in execution time. Overall, the `WITH`

query adds an additional JOIN, two extra sorts (quicksort and an external merge sort), group aggregation on all of the attributes, and the overhead of building a temporary view which is then also scanned.

Neo4j does not suffer from the same degradation – its internal planner performs *Eager Aggregation* of the query on the variables, filters on the predicates, and then sorts. The number of database accesses also *drops* by 17.27% with the use of `WITH`.

The execution plans for both of the queries executed on Neo4j and Postgres are contained in Appendix E.1 and E.2 to view.

4.3 Evaluation of initial success criteria

To conclude this chapter, I now look at how the tool built and the project overall compares against the original success criteria I outlined before starting any work, in addition to the requirements mentioned in §2.5.4.

The tool must be able to convert a wide range of Neo4j graph schemas to an equivalent relational database schema.

Although extensive testing has not been undertaken, for the most part, this requirement has been met. As mentioned in §3.1, it was necessary to make this part more parallel than initially intended, as it was simply too slow on the larger dump files.

The performance was also much better than I had initially predicted. The main bottleneck was in creating the *Edges* relation in Postgres, but compared to the time taken to create the graph from the same dump in Neo4j, it was faster.

The tool must be able to convert a modest subset of Cypher input to executable SQL automatically.

This requirement was successfully met in that all the keywords and graph functions I intended to convert could be translated.

This requirement, alongside the first one, also satisfy the first success criteria I set, which was to *produce a working translator tool from Cypher to SQL*. The automated workflow mentioned in my second success criteria has also been met, given that I have produced a suitably automated tool.

The tool must be able to show the execution times of running Cypher and SQL on Neo4j and Postgres respectively.

This requirement was met without difficulty. The tool itself does not yet track more advanced metrics, such as the number of database accesses, but these were computed for the Evaluation separately from the tool anyway.

The tool must have a module that collates the results from both Neo4j and Postgres into text files, which can be compared for accuracy and completeness.

With the exception of some encoding differences, this requirement was achieved. The tool can be set to record the outputs of the databases to both confirm the validity of the tool, and to make the tool more useful in the future as a possible addition to graph database systems, where a module would be needed to return the data from Postgres in the same format as Neo4j.

The tool should be coded to allow for extensions to the language to be included with minimal change to the whole code.

The current tool developed allowed me to implement my main required extension of increasing the capabilities of Cypher. More complex changes and refactoring would be needed to extend the tool further, such as allowing it to handle new keywords like `OPTIONAL`.

The experiments discussed in both §4.2.2 and §4.2.3 helped me to identify some weak points in the query optimiser of Neo4j. Adding labels to a Cypher query showed large performance gains in Neo4j, as it transpired the underlying engine uses labels to improve the execution plan of the query. This is an area still under current research.

My translation mechanism also performs some optimisations, such as attempting to find the best relations to join in queries.

Chapter 5

Conclusion

This project set out to evaluate thoroughly the performance of graph databases and their equivalent schema on relational databases. By successfully building a tool to translate Neo4j graphs and Cypher queries to Postgres and SQL respectively, the dissertation has been able to effectively evaluate the performance, scalability, and usability of both Neo4j and two different relational representations. Furthermore, an extension to Cypher was implemented to highlight the additional expressiveness of the relational representation.

5.1 Achievements

By conducting a series of experiments against both databases, I was able to identify areas where Neo4j was outperformed by Postgres, and where the inverse was true.

One fascinating conclusion was that for general graph queries and functions, Postgres performed considerably better, up to 2,500× faster when larger graphs were used. The addition of indexing to the Cypher query (through the use of labels on the nodes) allowed Neo4j to plan its execution better, and improve its performance considerably. However, Neo4j could not scale for some `shortestPath` queries when graphs contained more than 20,000 nodes, unlike Postgres which did not have this issue.

This performance gain was also achievable with small overheads, in terms of the space consumed by Postgres and the initial time to convert the graph schema. The tool even implemented some basic optimisations to the SQL translation unit where possible.

Where Neo4j performed particularly well was when the translation to SQL became harder and less obvious. Cypher queries containing the keyword `WITH` performed and scaled better than the SQL equivalent on Postgres.

Two different relational representations were built and tested, both with differing trade-offs. The adjacency list model overall was the optimal choice, due to the size and time complexities of building the transitive closure, and the minimal gains subsequently received from this representation. Both did allow for advanced graph queries and algorithms to automatically be translated to SQL, something that has not been done to my knowledge before.

The extension implemented added not only additional functionality to the tool, but showed how the adjacency list model developed could be used beyond the current capa-

bilities of Neo4j.

5.2 Further work

More investigation into possible database and query optimisations could be looked into if work on this project were to be continued. At the beginning of the project, Neo4j itself has migrated from version 3.0 to version 3.1, and I imagine future versions will only become more stable and contain further improvements.

The translator tool itself, as mentioned during the dissertation, is currently in a stable state, but one where extensions would be relatively difficult to add without some modification to parts of the code. Ideally, the code would be refactored to allow for easier alterations in the future, alongside changes to parts of the code where the translation could be improved. For example, Cypher statements containing `WITH` can only accept statements with a very rigid structure, and likewise, there are some known bugs to do with using aliases of return values.

This dissertation only focused on translating Cypher (executing on Neo4j) to SQL, which will be executed on Postgres. Additional studies could look into other graph query languages and databases, as mentioned briefly in §2.1.1.

Finally, I believe a *hybrid* version of the tool is very feasible and will contain multiple benefits. Choosing dynamically at runtime the optimal representation to run a Cypher query on, and return the exact same format of results, regardless of the internal processing undertaken, would be extremely useful. This would require more extensive testing of queries and database types, however.

5.3 Final remarks

With hindsight, more initial development would have gone into making the first steps of the implementation more formalised, so that it would have been easier to extend with more translation capabilities later on. Nonetheless, the automated tool developed helped me produce insightful results (summarised in §4) into how Neo4j and Postgres handled graph databases and graph queries. Finally, by meeting the original requirements I set out to achieve (summarised in §4.3), I believe this project has laid down the framework for continued work in this very active research area.

Bibliography

[1] A. Woodie, "5 Factors Driving the Graph Database Explosion." `https://www.datanami.com/2016/08/19/5-factors-driving-graph-database-explosion/`, August 2016.

[2] S. Edlich, "List of NoSQL Databases." `http://nosql-database.org/`.

[3] "Db-Engines Ranking." `http://db-engines.com/en/ranking` (last accessed: 22 March 2017).

[4] "Neo4j System Properties." `http://db-engines.com/en/system/Neo4j`.

[5] J. Holsch and M. Grossniklaus, "An Algebra and Equivalences to Transform Graph Patterns in Neo4j," in *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference (EDBT/ICDT 2016) (EDBT/ICDT)* (T. Palpanas, E. Pitoura, W. Martens, S. Maabout, and K. Stefanidis, eds.), no. 1558 in CEUR Workshop Proceedings, (Aachen), 2016.

[6] "The Database Model Showdown: An RDBMS vs. Graph Comparison." `https://neo4j.com/blog/database-model-comparison/`, November 2015.

[7] C. Ireland, D. Bowers, M. Newton, and K. Waugh, "A classification of object-relational impedance mismatch," in *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA'09. First International Conference on*, pp. 36–43, IEEE, 2009.

[8] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader, "A performance evaluation of open source graph databases," in *Proceedings of the first workshop on Parallel programming for analytics applications*, pp. 11–18, ACM, 2014.

[9] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. O'Reilly, 2015.

[10] M. A. Rodriguez, "The Benefits of the Gremlin Graph Traversal Machine," September 2015.

[11] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[12] K. Thulasiraman and M. Swamy, *Graphs theory and algorithms*. John Wiley and Sons, 2011.

[13] T. G. Griffin, "1A Databases lectures 5 and 6: (University of Cambridge - Computer Science Tripos," November 2016.

[14] A. Eisenberg and J. Melton, "Sql: 1999, formerly known as sql3," *ACM Sigmod record*, vol. 28, no. 1, pp. 131–138, 1999.

[15] D. R. Harris, D. W. Henderson, R. Kavuluru, A. J. Stromberg, and T. R. Johnson, "Using common table expressions to build a scalable boolean query generator for clinical data warehouses," *IEEE journal of biomedical and health informatics*, vol. 18, no. 5, pp. 1607–1613, 2014.

[16] "Postgresql: Documentation: 9.4: With queries (common table expressions)." `https://www.postgresql.org/docs/9.4/static/queries-with.html`.

[17] G. Dong, L. Libkin, J. Su, and L. Wong, "Maintaining transitive closure of graphs in sql," *International Journal of Information Technology*, vol. 51, no. 1, p. 46, 1999.

[18] I. Sommerville, *Software Engineering*. Pearson Education Limited, 10th ed., 2016.

[19] G. Melancon, "Just how dense are dense graphs in the real world?," *Proceedings of the 2006 AVI workshop on Beyond time and errors novel evaluation methods for information visualization - BELIV '06*, pp. 75–81, September 2006.

Appendix A

Example CTE framework

Listing A.1: General pattern of a CTE within an SQL statement [15].

```
WITH common_table_exp_a AS (
   SELECT unnest(rightnode) AS xx
   FROM adjList_from INNER JOIN nodes y ON leftnode = y.id
),
common_table_exp_b AS (
   SELECT unnest(rightnode) AS xx
   FROM adjList_from INNER JOIN common_table_exp_a ON leftnode = xx
),
common_table_exp_c AS (
   SELECT xx FROM common_table_exp_b
   UNION ALL
   SELECT xx FROM common_table_exp_a
)
SELECT count(*) FROM nodes n01 INNER JOIN common_table_exp_c ON xx = id
WHERE n01.label LIKE "%website%";
```

Appendix B

SQL translations of variable length path queries

Listing B.1: Variable length Cypher query.

```
MATCH (a:Owner {state:"Wyoming"})-[*1..3]->(b:Website {domain:"com"})
RETURN COUNT(b);
```

Listing B.2: Using the transitive closure.

```
CREATE TEMP VIEW zerostep AS SELECT id from nodes
WHERE state = "wyoming" AND label LIKE "%owner%";

CREATE TEMP VIEW step AS (
   WITH graphT AS (
      SELECT idr as x, idl as y FROM tClosure
      JOIN zerostep on idl = zerostep.id
      JOIN nodes as n on idr = n.id
      WHERE depth <= 3 AND depth >= 1
   )
   SELECT * from graphT
);

SELECT count(*) FROM nodes n01 JOIN step on x = n01.id
WHERE n01.label LIKE "%website%" AND n01.domain = "com";
```

Listing B.3: Using the adjacency list representation.

```
WITH a1 AS (
   SELECT unnest(rightnode) AS xx FROM adjList_from
   INNER JOIN nodes zz ON leftnode = zz.id
   WHERE zz.state = "wyoming" AND zz.label LIKE "%owner%"
),
b1 AS (
   SELECT unnest(rightnode) AS xx FROM adjList_from
   INNER JOIN a1 ON leftnode = xx
),
c1 AS (
   SELECT unnest(rightnode) AS xx FROM adjList_from
   INNER JOIN b1 ON leftnode = xx
),
d1 AS (
   SELECT xx FROM c1 UNION ALL SELECT xx FROM b1
   UNION ALL SELECT xx FROM a1
)
SELECT count(*) FROM nodes n01 INNER JOIN d1 ON xx = id
WHERE n01.domain = "com" AND n01.label LIKE "%website%";
```

Appendix C

SQL FOREACH function

```
1  CREATE FUNCTION doForEachFunc(int[], field TEXT, newV TEXT) RETURNS void
       AS $$
2      DECLARE
3        x int;
4        r record;
5        l text;
6      BEGIN
7        if array_length($1, 1) > 0 THEN
8          FOREACH x SLICE 0 IN ARRAY $1 LOOP
9            FOR r IN SELECT label from nodes where id = x LOOP
10             EXECUTE 'UPDATE nodes SET ' || field || '=' ||
                   quote_literal(newV) || ' WHERE id = ' || x;
11             l := replace(r.label, ', ', '_');
12             EXECUTE 'UPDATE ' || l || ' SET ' || field || '=' ||
                   quote_literal(newV) || ' WHERE id = ' || x;
13           END LOOP;
14         END LOOP;
15       END IF;
16     END;
17 $$ LANGUAGE plpgsql;
```

Appendix D

Execution plans from Experiment 3

D.1 Execution plans from Neo4j

```
1 MATCH p=shortestPath((f {name:"omega"})-[*1..3]->(t))
2 RETURN COUNT(t.node_id);
```

```
+-----------+
| count(t.node_id) |
+-----------+
| 2760             |
+-----------+
1 row
7254 ms

Compiler CYPHER 3.1

Planner COST

Runtime INTERPRETED

+-----------------------------------------------------------+
| Operator             Est. Rows      Rows     DB Hits      Variables              |
+-----------------------------------------------------------+
| +ProduceResults          1568         1           0      count(t.node_id)       |
| |                                                         |
| +EagerAggregation        1568         1        2760      count(t.node_id)       |
| |                                                         |
| +ShortestPath          457185      2760     1566412      anon[40], p -- f, t    |
| |                                                         |
| +CartesianProduct     2457185   1566412           0      f -- t                 |
| |\                                                        |
| | +AllNodesScan          4957   1566412     1566728      t                      |
| |                                                         |
| +Filter                 496       316        4957      f                      |
| |                                                         |
| +AllNodesScan          4957      4957        4958      f                      |
+-----------------------------------------------------------+

Total database accesses: 3145815
```

```
1 MATCH p=shortestPath((f {name:"omega"})-[*1..3]->(t:Local {ref_count:2}))
2 RETURN COUNT(t.node_id);
```

```
+-----------+
| count(t.node_id) |
+-----------+
| 10               |
+-----------+
1 row
1265 ms

Compiler CYPHER 3.1

Planner COST

Runtime INTERPRETED

+-----------------------------------------------------------------+
| Operator                 Est. Rows    Rows    DB Hits    Variables |
+-----------------------------------------------------------------+
| +ProduceResults                370       1          0    count(t.node_id) |
| |                                                                |
| +EagerAggregation              370       1         10    count(t.node_id) |
| |                                                                |
| +ShortestPath                136962      10       3160    anon[40], p -- f, t |
| |                                                              |
| +CartesianProduct            136962    3160          0    f -- t |
| |\                                                             |
| | +Filter                       276    3160     873108    t |
| | |                                                            |
| | +NodeByLabelScan             2763  873108     873424    t |
| |                                                              |
| +Filter                        496     316       4957    f |
| |                                                            |
| +AllNodesScan                 4957    4957       4958    f |
+-----------------------------------------------------------------+

Total database accesses: 1759617
```

D.2 Execution plans from Postgres

```
MATCH p=shortestPath((f {name:"omega"})-[*1..3]->(t))
RETURN COUNT(t.node_id);
```

```
WITH a AS(
   SELECT unnest(rightnode) AS xx, 1 AS Depth, ARRAY[id] AS Path, id AS
       Start
   FROM adjList_from INNER JOIN nodes q ON leftnode = id WHERE q.name =
       ARRAY["omega"]
),
b AS (
   SELECT unnest(rightnode) AS xx, 2 AS Depth, a.Path || ARRAY[xx] AS
       Path, a.Path[1] AS Start
   FROM adjList_from INNER JOIN a ON leftnode = xx
),
c AS (
   SELECT unnest(rightnode) AS xx, 3 AS Depth, b.Path || ARRAY[xx] AS
       Path, b.Path[1] AS Start
   FROM adjList_from INNER JOIN b ON leftnode = xx
),
d AS (
   SELECT * FROM a UNION ALL SELECT * FROM b UNION ALL SELECT * FROM c
),
finStep AS (
   SELECT n01.node_id, min(Depth), xx, Start FROM nodes n01 INNER JOIN d
       ON xx = id
   GROUP BY node_id, xx, Start
)

SELECT count(n01.node_id) FROM finStep n01;
```

```
Aggregate  (cost=83989473.17..83989473.18 rows=1 width=4) (actual time=12.531..12.531 rows=1 loops=1)
  CTE a
    ->  Hash Join  (cost=132.91..375.70 rows=28600 width=31) (actual time=0.835..1.780 rows=460 loops=1)
          Hash Cond: (adjlist_from.leftnode = q.id)
          ->  Seq Scan on adjlist_from  (cost=0.00..80.83 rows=4483 width=31) (actual time=0.005..0.291 rows=4483 loops=1)
          ->  Hash  (cost=128.96..128.96 rows=316 width=4) (actual time=0.816..0.816 rows=316 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 12kB
                ->  Seq Scan on nodes q  (cost=0.00..128.96 rows=316 width=4) (actual time=0.006..0.779 rows=316 loops=1)
                      Filter: (name = '{omega}'::text[])
                      Rows Removed by Filter: 4641
  CTE b
    ->  Hash Join  (cost=136.87..15509.37 rows=2860000 width=63) (actual time=1.003..1.971 rows=1369 loops=1)
          Hash Cond: (a.xx = adjlist_from_1.leftnode)
          ->  CTE Scan on a  (cost=0.00..572.00 rows=28600 width=36) (actual time=0.000..0.037 rows=460 loops=1)
          ->  Hash  (cost=80.83..80.83 rows=4483 width=31) (actual time=0.992..0.992 rows=4483 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 293kB
                ->  Seq Scan on adjlist_from adjlist_from_1  (cost=0.00..80.83 rows=4483 width=31) (actual time=0.001..0.383
     rows=4483 loops=1)
  CTE c
    ->  Hash Join  (cost=136.87..1537386.87 rows=286000000 width=63) (actual time=0.995..2.219 rows=1691 loops=1)
          Hash Cond: (b.xx = adjlist_from_2.leftnode)
          ->  CTE Scan on b  (cost=0.00..57200.00 rows=2860000 width=36) (actual time=0.001..0.126 rows=1369 loops=1)
          ->  Hash  (cost=80.83..80.83 rows=4483 width=31) (actual time=0.988..0.988 rows=4483 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 293kB
                ->  Seq Scan on adjlist_from adjlist_from_2  (cost=0.00..80.83 rows=4483 width=31) (actual time=0.002..0.382
     rows=4483 loops=1)
```

```
  CTE d
    -> Append (cost=0.00..5777772.00 rows=288888600 width=44) (actual time=0.837..7.000 rows=3520 loops=1)
          -> CTE Scan on a a_1 (cost=0.00..572.00 rows=28600 width=44) (actual time=0.836..1.891 rows=460 loops=1)
          -> CTE Scan on b b_1 (cost=0.00..57200.00 rows=2860000 width=44) (actual time=1.004..2.301 rows=1369 loops=1)
          -> CTE Scan on c (cost=0.00..5720000.00 rows=286000000 width=44) (actual time=0.995..2.632 rows=1691 loops=1)
  CTE finstep
    -> GroupAggregate (cost=66603221.73..72197129.23 rows=198280000 width=16) (actual time=10.840..11.667 rows=2760
    loops=1)
          Group Key: n01_1.node_id, d.xx, d.start
          -> Sort (cost=66603221.73..67325443.23 rows=288888600 width=16) (actual time=10.838..10.983 rows=3520 loops=1)
                Sort Key: n01_1.node_id, d.xx, d.start
                Sort Method: quicksort Memory: 262kB
                -> Hash Join (cost=178.53..11194611.78 rows=288888600 width=16) (actual time=2.040..9.730 rows=3520 loops=1)
                      Hash Cond: (d.xx = n01_1.id)
                      -> CTE Scan on d (cost=0.00..5777772.00 rows=288888600 width=12) (actual time=0.838..7.489 rows=3520
    loops=1)
                      -> Hash (cost=116.57..116.57 rows=4957 width=8) (actual time=1.198..1.198 rows=4957 loops=1)
                            Buckets: 1024 Batches: 1 Memory Usage: 194kB
                            -> Seq Scan on nodes n01_1 (cost=0.00..116.57 rows=4957 width=8) (actual time=0.001..0.652
    rows=4957 loops=1)
  -> CTE Scan on finstep n01 (cost=0.00..3965600.00 rows=198280000 width=4) (actual time=10.842..12.400 rows=2760 loops=1)
 Planning time: 0.676 ms
 Execution time: 12.751 ms
(44 rows)
```

```
# obtained from https://explain.depesz.com/s/kyyg
# online tool for visualising Postgres EXPLAIN ANALYZE outputs

Per node type stats
node type          count    sum of times       % of query
Aggregate          1        0.131 ms           1.0 %
Append             1        0.176 ms           1.4 %
CTE Scan           7        26.876 ms          214.5 %
GroupAggregate     1        0.684 ms           5.5 %
Hash               4        1.798 ms           14.3 %
Hash Join          4        3.763 ms           30.0 %
Seq Scan           5        2.487 ms           19.8 %
Sort               1        1.253 ms           10.0 %

Per table stats
Table name         Scan count     Total time      % of query
scan type          count          sum of times    % of table
adjlist_from       3              1.056 ms        8.4 %
Seq Scan           3              1.056 ms        100.0 %
nodes              2              1.431 ms        11.4 %
Seq Scan           2              1.431 ms        100.0 %
```

```
MATCH p=shortestPath((f {name:"omega"})-[*1..3]->(t:Local {ref_count:2}))
RETURN COUNT(t.node_id);
```

```
WITH a AS(
   SELECT unnest(rightnode) AS xx, 1 AS Depth, ARRAY[id] AS Path, id AS
       Start
   FROM adjList_from INNER JOIN nodes q ON leftnode = id WHERE q.name =
       ARRAY["omega"]
),
b AS (
   SELECT unnest(rightnode) AS xx, 2 AS Depth, a.Path || ARRAY[xx] AS
       Path, a.Path[1] AS Start
   FROM adjList_from INNER JOIN a ON leftnode = xx
),
c AS (
   SELECT unnest(rightnode) AS xx, 3 AS Depth, b.Path || ARRAY[xx] AS
       Path, b.Path[1] AS Start
   FROM adjList_from INNER JOIN b ON leftnode = xx
),
d AS (
   SELECT * FROM a UNION ALL SELECT * FROM b UNION ALL SELECT * FROM c
),
finStep AS (
   SELECT n01.node_id, min(Depth), xx, Start FROM nodes n01 INNER JOIN d
       ON xx = id
   WHERE label LIKE "%local%" AND n01.ref_count = "2"
   GROUP BY node_id, xx, Start
)

SELECT count(n01.node_id) FROM finStep n01;
```

```
Aggregate  (cost=14246132.65..14246132.66 rows=1 width=4) (actual time=8.631..8.631 rows=1 loops=1)
  CTE a
    ->  Hash Join  (cost=132.91..375.70 rows=28600 width=31) (actual time=0.850..1.762 rows=460 loops=1)
          Hash Cond: (adjlist_from.leftnode = q.id)
          ->  Seq Scan on adjlist_from  (cost=0.00..80.83 rows=4483 width=31) (actual time=0.005..0.287 rows=4483 loops=1)
          ->  Hash  (cost=128.96..128.96 rows=316 width=4) (actual time=0.832..0.832 rows=316 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 12kB
                ->  Seq Scan on nodes q  (cost=0.00..128.96 rows=316 width=4) (actual time=0.006..0.786 rows=316 loops=1)
                      Filter: (name = '{omega}'::text[])
                      Rows Removed by Filter: 4641
  CTE b
    ->  Hash Join  (cost=136.87..15509.37 rows=2860000 width=63) (actual time=1.001..1.925 rows=1369 loops=1)
          Hash Cond: (a.xx = adjlist_from_1.leftnode)
          ->  CTE Scan on a  (cost=0.00..572.00 rows=28600 width=36) (actual time=0.000..0.063 rows=460 loops=1)
          ->  Hash  (cost=80.83..80.83 rows=4483 width=31) (actual time=0.989..0.989 rows=4483 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 293kB
                ->  Seq Scan on adjlist_from adjlist_from_1  (cost=0.00..80.83 rows=4483 width=31) (actual time=0.001..0.386
     rows=4483 loops=1)
  CTE c
    ->  Hash Join  (cost=136.87..1537386.87 rows=286000000 width=63) (actual time=1.004..2.187 rows=1691 loops=1)
          Hash Cond: (b.xx = adjlist_from_2.leftnode)
          ->  CTE Scan on b  (cost=0.00..57200.00 rows=2860000 width=36) (actual time=0.000..0.103 rows=1369 loops=1)
          ->  Hash  (cost=80.83..80.83 rows=4483 width=31) (actual time=0.992..0.992 rows=4483 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 293kB
                ->  Seq Scan on adjlist_from adjlist_from_2  (cost=0.00..80.83 rows=4483 width=31) (actual time=0.002..0.374
     rows=4483 loops=1)
  CTE d
    ->  Append  (cost=0.00..5777772.00 rows=288888600 width=44) (actual time=0.851..6.912 rows=3520 loops=1)
```

```
          ->  CTE Scan on a a_1  (cost=0.00..572.00 rows=28600 width=44) (actual time=0.851..1.880 rows=460 loops=1)
          ->  CTE Scan on b b_1  (cost=0.00..57200.00 rows=2860000 width=44) (actual time=1.002..2.242 rows=1369 loops=1)
          ->  CTE Scan on c  (cost=0.00..5720000.00 rows=286000000 width=44) (actual time=1.004..2.608 rows=1691 loops=1)
   CTE finstep
     ->  GroupAggregate  (cost=6902917.79..6909688.72 rows=240000 width=16) (actual time=8.623..8.626 rows=10 loops=1)
           Group Key: n01_1.node_id, d.xx, d.start
           ->  Sort  (cost=6902917.79..6903791.98 rows=349674 width=16) (actual time=8.621..8.621 rows=10 loops=1)
                 Sort Key: n01_1.node_id, d.xx, d.start
                 Sort Method: quicksort  Memory: 25kB
                 ->  Hash Join  (cost=141.43..6864742.42 rows=349674 width=16) (actual time=1.810..8.606 rows=10 loops=1)
                       Hash Cond: (d.xx = n01_1.id)
                       ->  CTE Scan on d  (cost=0.00..5777772.00 rows=288888600 width=12) (actual time=0.851..7.391 rows=3520
     loops=1)
                       ->  Hash  (cost=141.36..141.36 rows=6 width=8) (actual time=0.910..0.910 rows=10 loops=1)
                             Buckets: 1024  Batches: 1  Memory Usage: 1kB
                             ->  Seq Scan on nodes n01_1  (cost=0.00..141.36 rows=6 width=8) (actual time=0.039..0.908 rows=10
     loops=1)
                                   Filter: ((label ~~ '%local%'::text) AND (ref_count = 2))
                                   Rows Removed by Filter: 4947
   ->  CTE Scan on finstep n01  (cost=0.00..4800.00 rows=240000 width=4) (actual time=8.624..8.630 rows=10 loops=1)
 Planning time: 0.687 ms
 Execution time: 8.831 ms
(46 rows)
```

```
# obtained from https://explain.depesz.com/s/kyyg
# online tool for visualising Postgres EXPLAIN ANALYZE outputs

Per node type stats
node type          count    sum of times       % of query
Aggregate          1        0.001 ms           0.0 %
Append             1        0.182 ms           2.1 %
CTE Scan           7        22.917 ms          265.5 %
GroupAggregate     1        0.005 ms           0.1 %
Hash               4        1.269 ms           14.7 %
Hash Join          4        2.913 ms           33.8 %
Seq Scan           5        2.741 ms           31.8 %
Sort               1        0.015 ms           0.2 %

Per table stats
Table name     Scan count     Total time         % of query
scan type      count          sum of times       % of table
adjlist_from   3              1.047 ms           12.1 %
Seq Scan       3              1.047 ms           100.0 %
nodes          2              1.694 ms           19.6 %
Seq Scan       2              1.694 ms           100.0 %
```

Appendix E

Execution plans from Experiment 4

E.1 Execution plans from Neo4j

NOTE: the execution plans have been modified so that they can fit in this document. Information pertaining to the variables has been omitted – these text files are just showing the execution plan and the corresponding number of database hits for each step.

```
1 MATCH (a:Global)-[m]->(b:Local) WITH a, COUNT(m) AS Glo_Loc_Count
2 WHERE Glo_Loc_Count >= 2
3 RETURN a.node_id, Glo_Loc_Count ORDER BY Glo_Loc_Count DESC;
```

```
Compiler CYPHER 3.1

Planner COST

Runtime INTERPRETED

+-----------------------------------+
| Operator            Est. Rows     Rows    DB Hits |
+-----------------------------------+
| +ProduceResults           412    33658          0 |
| |                                                 |
| +Projection               412    33658          0 |
| |                                                 |
| +Sort                     412    33658          0 |
| |                                                 |
| +Projection               412    33658      33658 |
| |                                                 |
| +Filter                   412    33658          0 |
| |                                                 |
| +Projection               549    66448          0 |
| |                                                 |
| +EagerAggregation         549    66448          0 |
| |                                                 |
| +Filter                301567   206251     301567 |
| |                                                 |
| +Expand(All)           301567   301567     396626 |
| |                                                 |
| +NodeByLabelScan        95059    95059      95060 |
+-----------------------------------+

Total database accesses: 826911
```

```
1 MATCH (a:Global)-[m]->(b:Local)
2 RETURN a.node_id;
```

```
Compiler CYPHER 3.1

Planner COST

Runtime INTERPRETED

+-----------------------------------------+
| Operator          Est. Rows  Rows  DB Hits |
+-----------------------------------------+
| +ProduceResults       301567  206251       0 |
| |                                           |
| +Projection           301567  206251  206251 |
| |                                           |
| +Filter               301567  206251  301567 |
| |                                           |
| +Expand(All)          301567  301567  396626 |
| |                                           |
| +NodeByLabelScan       95059   95059   95060 |
+-----------------------------------------+

Total database accesses: 999504
```

E.2 Execution plans from Postgres

```
MATCH (a:Global)-[m]->(b:Local) WITH a, COUNT(m) AS Glo_Loc_Count
WHERE Glo_Loc_Count >= 2
RETURN a.node_id, Glo_Loc_Count ORDER BY Glo_Loc_Count DESC;
```

```
CREATE TEMP VIEW wA AS (
   WITH a AS (
      SELECT n1.id AS a1, n2.id AS a2, e1.* FROM global n1
      INNER JOIN edges e1 on n1.id = e1.idl
      INNER JOIN local n2 on e1.idr = n2.id
   )
   SELECT n01.*, count(a.*) AS glo_loc_count FROM nodes n01, a
   WHERE n01.id = a.a1 GROUP BY n01.name, n01.opus_lite, n01.mono_time,
       n01.pid, n01.sys_time, n01.type, n01.status, n01.id, n01.timestamp,
       n01.node_id, n01.ref_count, n01.label, n01.value
);

SELECT node_id FROM wA
WHERE glo_loc_count >= 2 ORDER BY glo_loc_count desc;
```

```
Sort  (cost=193354.29..194135.20 rows=312363 width=12) (actual time=3960.977..3962.307 rows=33658 loops=1)
  Sort Key: wa.glo_loc_count
  Sort Method: quicksort  Memory: 2944kB
  -> Subquery Scan on wa  (cost=139983.01..159505.69 rows=312363 width=12) (actual time=2893.078..3955.324 rows=33658
     loops=1)
       -> GroupAggregate  (cost=139983.01..156382.06 rows=312363 width=117) (actual time=2893.077..3952.764 rows=33658
     loops=1)
            Group Key: n01.name, n01.opus_lite, n01.mono_time, n01.pid, n01.sys_time, n01.type, n01.status, n01.id,
     n01."timestamp", n01.node_id, n01.ref_count, n01.label, n01.value
            Filter: (count(a.*) >= 2)
            Rows Removed by Filter: 32790
            CTE a
              -> Hash Join  (cost=13797.85..49440.39 rows=312363 width=30) (actual time=80.016..487.095 rows=206251
     loops=1)
                   Hash Cond: (e1.idr = n2.id)
                   -> Hash Join  (cost=3356.83..28206.02 rows=312363 width=26) (actual time=18.810..271.382 rows=301567
     loops=1)
                        Hash Cond: (e1.idl = n1.id)
                        -> Seq Scan on edges e1  (cost=0.00..13605.28 rows=812028 width=22) (actual time=0.016..54.204
     rows=812028 loops=1)
                        -> Hash  (cost=2168.59..2168.59 rows=95059 width=4) (actual time=18.771..18.771 rows=95059
     loops=1)
                             Buckets: 16384  Batches: 1  Memory Usage: 3342kB
                             -> Seq Scan on global n1  (cost=0.00..2168.59 rows=95059 width=4) (actual time=0.003..9.834
     rows=95059 loops=1)
                   -> Hash  (cost=6012.12..6012.12 rows=269912 width=4) (actual time=61.174..61.174 rows=269912 loops=1)
                        Buckets: 16384  Batches: 4  Memory Usage: 2383kB
                        -> Seq Scan on local n2  (cost=0.00..6012.12 rows=269912 width=4) (actual time=0.001..29.543
     rows=269912 loops=1)
            -> Sort  (cost=90542.62..91323.52 rows=312363 width=117) (actual time=2892.827..3867.140 rows=206251 loops=1)
                 Sort Key: n01.name, n01.opus_lite, n01.mono_time, n01.pid, n01.sys_time, n01.type, n01.status, n01.id,
     n01."timestamp", n01.node_id, n01.ref_count, n01.label, n01.value
                 Sort Method: external merge  Disk: 28600kB
                 -> Hash Join  (cost=20190.00..42816.52 rows=312363 width=117) (actual time=187.800..825.581 rows=206251
     loops=1)
                      Hash Cond: (a.a1 = n01.id)
                      -> CTE Scan on a  (cost=0.00..6247.26 rows=312363 width=32) (actual time=80.025..559.427
     rows=206251 loops=1)
                      -> Hash  (cost=9330.00..9330.00 rows=400000 width=89) (actual time=107.492..107.492 rows=400000
     loops=1)
                           Buckets: 4096  Batches: 16  Memory Usage: 2652kB
                           -> Seq Scan on nodes n01  (cost=0.00..9330.00 rows=400000 width=89) (actual time=0.015..30.736
     rows=400000 loops=1)
Planning time: 0.218 ms
Execution time: 3968.535 ms
(31 rows)
```

```
MATCH (a:Global)-[m]->(b:Local)
RETURN a.node_id;
```

```
WITH a AS (
   SELECT n1.id AS a1, n2.id AS a2, e1.*
   FROM nodes n1 INNER JOIN edges e1 on n1.id = e1.idl
   INNER JOIN nodes n2 on e1.idr = n2.id
   WHERE n1.label LIKE "%local%" AND n2.label LIKE "%local%"
),
b AS (
   SELECT n1.id AS b1, n2.id AS b2, e2.*
   FROM nodes n1 INNER JOIN edges e2 on n1.id = e2.idl
   INNER JOIN nodes n2 on e2.idr = n2.id
   WHERE n1.label LIKE "%local%" AND n2.label LIKE "%process%"
),
c AS (
   SELECT n1.id AS c1, n2.id AS c2, e3.*
   FROM nodes n1 INNER JOIN edges e3 on n1.id = e3.idr
   INNER JOIN nodes n2 on e3.idl = n2.id
   WHERE n1.label LIKE "%process%" AND n2.label LIKE "%local%"
)

SELECT count(n01.*) FROM nodes n01, a, b, c
WHERE a.a2 = b.b1 AND b.b2 = c.c1 AND n01.id = a.a1 AND a.a1 != b.b2 AND
    a.a2 != c.c2 AND a.a1 = c.c2;
```

```
Hash Join  (cost=65333.39..81832.91 rows=312363 width=4) (actual time=192.700..771.110 rows=206251 loops=1)
  Hash Cond: (a.a1 = n01.id)
  CTE a
    ->  Hash Join  (cost=13797.85..49440.39 rows=312363 width=30) (actual time=84.375..493.424 rows=206251 loops=1)
          Hash Cond: (e1.idr = n2.id)
          ->  Hash Join  (cost=3356.83..28206.02 rows=312363 width=26) (actual time=21.228..269.117 rows=301567 loops=1)
                Hash Cond: (e1.idl = n1.id)
                ->  Seq Scan on edges e1  (cost=0.00..13605.28 rows=812028 width=22) (actual time=0.007..53.361 rows=812028
    loops=1)
                ->  Hash  (cost=2168.59..2168.59 rows=95059 width=4) (actual time=21.177..21.177 rows=95059 loops=1)
                      Buckets: 16384  Batches: 1  Memory Usage: 3342kB
                      ->  Seq Scan on global n1  (cost=0.00..2168.59 rows=95059 width=4) (actual time=0.006..10.731 rows=95059
    loops=1)
          ->  Hash  (cost=6012.12..6012.12 rows=269912 width=4) (actual time=63.087..63.087 rows=269912 loops=1)
                Buckets: 16384  Batches: 4  Memory Usage: 2383kB
                ->  Seq Scan on local n2  (cost=0.00..6012.12 rows=269912 width=4) (actual time=0.002..30.558 rows=269912
    loops=1)
  ->  CTE Scan on a  (cost=0.00..6247.26 rows=312363 width=4) (actual time=84.378..557.889 rows=206251 loops=1)
  ->  Hash  (cost=9330.00..9330.00 rows=400000 width=8) (actual time=108.173..108.173 rows=400000 loops=1)
        Buckets: 16384  Batches: 4  Memory Usage: 3926kB
        ->  Seq Scan on nodes n01  (cost=0.00..9330.00 rows=400000 width=8) (actual time=0.012..58.548 rows=400000 loops=1)
Planning time: 0.379 ms
Execution time: 777.877 ms
(20 rows)
```

Appendix F

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Translating Cypher: an investigation in applying graph queries to relational databases.

O. J. Crawford, Trinity Hall
Originator: Dr. Ripduman Sohan
21st October 2016

Project Supervisor: Mr. Lucian Carata
Director of Studies: Prof. Simon Moore
Project Overseers: Prof. Marcelo Fiore & Prof. Ian Leslie

Introduction

Neo4J is a native graph database built to store and process both data, and the relationships between them. It is queried by using the Cypher language, which was developed by the same people as Neo4J. According to the developers of the language, it often requires 10x to 100x less code than SQL, which is the traditional language for querying relational data.[1] Thus, Cypher is extremely expressive and succinct for querying data with graph based properties.[2]

Neo4J however is not the panacea of graph databases. Relative to its relational counterparts, it can perform quite slowly for certain graph structures and query types. For example, dense graphs where the number of edges is almost maximal for the graph. I propose for my project a quantitative investigation into the differences between Neo4J and a relational database technology.

The rationale behind the project is the idea that the decades of work and optimisations dedicated to the field of relational databases makes them faster for performing queries.

The schema of relational database is also based on indexing for faster searching, whereas Neo4J uses index-free adjacency, which can be slower.

The main work of the project is to implement a tool that will perform automatic translation of both graph schemas and graph queries in Neo4J and Cypher respectively, to a relational schema and relational query language (SQL). From this point onwards, the tool will be known as *Cyp2SQL*. It will consist of a number of modules built during the project, including a schema converter from Neo4J to a relational database, a Cypher parsing unit, and the query translator itself. A module for comparing the outputs from both Neo4J and the relational DB will also need to be integrated.

I also propose a further investigation into both the tool itself, and the potential extensions as a result of the tool. The language of Cypher itself could be extended to increase its expressibility, as a result of the mapping to a relational database. A study of the effect of scalability will also be conducted. Moreover, *Cyp2SQL* will at the core of its unit rely on a translation similar to that of extending the relational algebra to graph structures. Potential optimisations for the queries may be possible here, but for the time being, will be considered as an extension to the overall project.

Starting point

As far as I am aware this project has not been attempted at Tripos level before. Prior to the project, I have had no experience of either graph databases or the Cypher language. Before writing the proposal however, I did investigate aspects of the Cypher language, to help familiarise myself slightly and make sure the project had potential.

Previous courses and experience that will help me include Databases and Compilers from IB, as well as experience in both Java and SQL from academia and industry. Software engineering practises I have learnt will also be crucial.

I have also been recommended, as a starting point for transforming graph patterns, the following paper[3]. Revising relational algebra followed on from this.

Resources required

For the initial phases of the project, it suffices for me to use my own laptop. It will run Windows 10, with the following specification:

- i5 dual-core processor (2.5 GHz)
- 8GB RAM
- 500GB SSD storage
- Java SDK, IntelliJ IDE, Neo4J Community Edition, Postgres 9.6 installed

I accept full responsibility for my machine and I have made contingency plans to protect myself against hardware/software failures.

I plan to use the Git version control system with a GitHub repository for the code. Further backups of the code and relevant documents will occur, to both the university's MCS system and Microsoft OneDrive.

Additionally, I plan to obtain some large datasets and more powerful computers for testing. Some of the data can be obtained online as open-source, but the Computer Lab also has some very large datasets (OPUS tool), which will be useful to evaluate against. As these datasets will be much larger than those on my laptop, some decent server machines will be needed to run the evaluation. The availability of DTG resources will be provided closer to the evaluation stage of the project. The originator and supervisor of the project have mentioned that this will be available for me.

Substance and Structure of the project

The project can be split into three parts:

1. the generation of the database schema
2. the translation of the queries
3. the optimisation/evaluation of the tool

Generation of the database schema

This will involve taking an existing graph database structure and generating an appropriate yet concise representation in relational form. In other words, manipulating the nodes and edges of a graph into entities and constraints of a relational database. It is important that all of the data contained within the graph is transferred over, such as the labels and properties. It may be that an additional file is created with some metadata inside.

An extension to this work could be comparing the performance increase (if any) of differing graph structures. It will also be important to evaluate the output of the schema translator, for example, comparing space overheads between graph and relational schemas.

Translation of the queries

This encompasses the majority of the work to be undertaken. It will roughly involve the following steps:

- Parsing Cypher using an appropriate parsing tool such as ANTLR, or modifying an existing tool for parsing the language (such as Neo4J's internal parsing module).
- The generation of tokens/a parse tree, which will then be translated to an intermediate representation. Some operations will map to SQL quite easily, such as SKIP and WHERE, whilst others will require a more complex translation. These include native graph operations that can easily be described in Cypher, but would require a much more complex SQL statement. For example, SQL Common Table Expressions (CTEs) and Recursive SQL queries.

- The output from the intermediate representation of the graph query then needs to be matched with the relational schema to generate an executable SQL statement. This is a crucial step of the tool, as it will also integrate possible query optimisations, and if designed correctly, could allow for very reusable code to other query languages if desired.

- The statement is then to be executed on a relational database. For the project, I plan to use Postgres, but the choice is fairly flexible.

My initial plan is to focus on a relatively large subset of the Cypher queries, particularly those that read the data (i.e. an OLAP[1] database). If time permits I will look into queries that can create and update data as well. The project will be a success if a large number of traditional graph queries can be represented using relational technology.

Optimisation and Correctness

After the core part of the translation tool is complete, I hope to look at optimisations of both the queries and the database setup. Topics to look at will include indexing, and reformatting queries to potentially improve their performance.

An important module for evaluation purposes will be the results unit, which will directly compare the outputs of Neo4J and the relational database. Otherwise, the evaluation will be a lot harder and less convincing if additional human input is required to show that the tool correctly transforms the queries.

Success criteria

The following are my criteria for the success of this project:

- Produce a working translator tool from the Cypher language to SQL (*Cyp2SQL*), that can be executed on a relational backend database. It should cover the majority of the read only queries of Cypher - to be more specific, the tool should be able to automatically translate most, if not all relationship structures, and be coded in a way that allows for extensions to be added if required.

- The tool should be as automated as possible. The workflow should be: the user has an existing graph database which is the input to the tool, the tool converts the schema, and then the user can either type in queries or have a batch of queries which are then translated.

- An evaluation of the performance between Neo4J and Postgres, in terms of either time, or a more accurate measurement such as the number of database accesses. A quantitative evaluation of the scalability properties of the tool should also have happened. The implementation should match results returned from both Neo4J

[1] Online Analytical Processing, which is optimised for query processing and database reads. Normal form is not important.

and Postgres as closely as possible. A further evaluation will be to study just how concise Cypher really is when compared to its equivalent SQL statements.

- A discussion on the optimisations discovered/implemented: this includes any potential query optimisations (using a form of relational algebra) and database optimisations (indexing).

Possible extensions

The core of the project is the translator module, but that itself can lead to further extensions. One has already been briefly mentioned, to do with optimising the whole process. This could be extended even more by creating a 'hybrid' system of both graph and relational technology, depending on factors such as the type of query or how scalable the performance will be. The concept will be that the user will input the query as normal, but the internal process will then at runtime decide the best strategy for the query.

I plan to only translate a large but important subset of Cypher. An extension would be a complete and formally tested translator for all queries possible in the language, or a smaller extension into comparing how the performance of updating the database is between Neo4J and Postgres.

Timetable and Milestones

Start Date – 24/10/16
Dissertation Due - 19/05/17

Weeks 1-2 *(24th October – 6th November)*

Initial background reading into the less familiar parts of the project, including the relational algebra, parsing tools, possible optimisations in graph queries etc.

Milestone 1: notes made from the reading material, and an initial plan for how to translate queries from Cypher to the intermediate algebra, and then finally to SQL.

Weeks 3-4 *(7th November – 20th November)*

Investigate and build a prototype parsing tool for Cypher. Further investigation into schema translations and more complex query translations. Implement small prototype which integrates Neo4J, Postgres, and hard-coded translation unit.

Milestone 2: prototype of parses and general framework for the tool. At this point, should have a firm idea of the specification on how to translate both the schema and the queries.

Weeks 5-6 *(21st November – 4th December)*

Implement relational schema generation module and simple translation unit.

Milestone 3: *Cyp2SQL* v1.0. (schema translator and simple translation unit). All preliminary reading completed in time for Christmas vacation.

Weeks 7-11 *(5th December – 8th January)*

Implement the majority of the tool, including the core translator unit for a large subset of the Cypher language. The module for comparing the results from both Neo4J and Postgres should also be developed. Towards the end of this milestone, prepare and discuss if necessary testing strategies and possibility of obtaining test data from the Computer Lab.

Milestone 4: *Cyp2SQL* v2.0. (automated translator unit in place with schema generation, not properly tested though). Draft testing strategy in place.

Weeks 12-13 *(9th January – 22nd January)*

Period to catch up on any missed work or unforeseen delays in the project.

Milestone 5: No milestone.

Weeks 14-15 *(23rd January – 5th February)*

Read up on possible optimisation techniques, recording them as appropriate. Testing on the main code should also begin. The progress report and presentation should be written.

Milestone 6: *Cyp2SQL* v3.0. (translator tool extended, bug fixes, refactoring if needed). Progress report handed in and presentation delivered.

Weeks 16-17 *(6th February – 19th February)*

Begin evaluation of the project using the test data. Evaluate with and without potential optimisations.

Milestone 7: Draft results from the tool. Discuss with supervisor plans for the dissertation (to be written in the next milestone).

Weeks 18-19 *(20th February – 5th March)*

Begin writing the dissertation. Release and demo the final tool.

Milestone 8: *Cyp2SQL* v4.0. (final code deliverable). Final results should now be collated.

Weeks 20-24 *(6th March – 9th April)*

Continue with the writing of the dissertation over Easter vacation.

Milestone 9: No milestone.

Weeks 25-27 *(10th April – 30th April)*

Further evaluation and complete draft version of the dissertation.

Milestone 10: Draft dissertation complete.

Weeks 28-30 *(1st May – 19th May)*

Proof reading and final additions completed, preferably 14 days before the submission date.

Milestone 11: Dissertation handed in for examination.

Bibliography

[1] http://info.neo4j.com/rs/neotechnology/images/Product%20At-A-Glance.pdf?_ga=1.198321833.720960388.1474885216

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

[3] Hölsch, Jürgen and Michael Grossniklaus. An Algebra and Equivalences to Transform Graph Patterns in Neo4j. EDBT/ICDT Workshops (2016).