Cyp2SQL – Documentation

Oliver Crawford
ojc37@cam.ac.uk

August 30, 2017

Abstract

Cyp2SQL is a tool for translating graph schemas and the graph query language Cypher to both a different database schema, and different database language (the original aim of this project was to translate Cypher to SQL). The tool can either be used as a standalone application, or, by utilising the .jar of the Java application, it can be used as an API for other applications. The code is licensed under Apache License, Version 2.0 (ALv2)

Contents

1 Overview

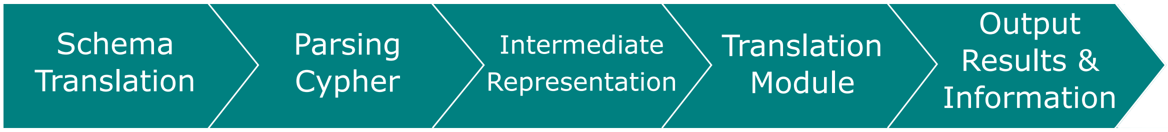


Figure 1: The five stages of the pipeline.

The pipeline of the tool consists of 5 stages (as shown in Figure 1):

- A *Schema Conversion* module for creating the initial database in a representation other than Neo4j (such as a relational schema in Postgres)
- A *Parsing Cypher* module, which is the first part of the translation to another database language (such as SQL). This module relies on the ANTLR framework, and other hand-written parsing code
- An *Intermediate Representation* is then built in Java – the result of this stage of the pipeline is a `DecodedQuery` object.
- A *Translation Module* takes this intermediate representation, and converts it to the target database language
- The *Output Results & Information* module can perform many different tasks, depending on the configuration desired:
  - It can execute both the original Cypher on Neo4j, and the newly created query on the target database, and record the execution times of the queries. The tool may also be made to write the results to local files, so that the outputs can be compared for accuracy.
  - The tool can also just execute the translated query on the database engine, and parse the results back into the *Cyp2SQL* tool.

The source code is entirely Java and is built using Maven. There are some additional features provided through the ANTLR tool, as described in §5. There are some script/.bat files that help the tool execute SQL on Postgres and pipe the results back into the *Cyp2SQL* tool.

1.1 Quick usage

Using the `cyp2sql-next.jar` file from the command line:

```
<-schema|-translate|-s|-t> <propertiesFile> <targetDatabaseName> <-dp|-dn|-r>
```

`-s | -schema` Runs the tool for schema conversion. See §3.

`-t | -translate` Runs the tool for query translation. See §4.

`propertiesFile` Path of the properties file `c2s_props.properties`. See §2.4.

`targetDatabaseName` Name of the relational database for either the converted graph schema to be stored into, or for the translated queries to execute on.

`dp | -dn | -r` **Only needed when in query translation mode**. See §4.1 for further details.

Using the code within another application as a library is also possible. Example usage:

```java
import database.postgres.InsertSchemaPostgres;
import intermediate_rep.DecodedQuery;
import production.C2SMain;
import production.C2SProperties;
import schema_conversion.SchemaConvert;

import java.io.IOException;

public class C2STestUsage {
public static void main(String args[]) {
// these need to be set before anything can occur.
String propsFile = "C:/Users/ocraw/IdeaProjects/cyp2sql-next/c2s_props.properties";
C2SProperties props = new C2SProperties(propsFile);

// name of the blank database to either:
// - convert the schema too
// - execute the translated Cypher on
String dbName = "testb";

String thingToDo = "translate";
switch (thingToDo) {
case "convert":
// the properties file must be edited first.
// a dump from Neo4j is also needed.
boolean successConvert = SchemaConvert.translate(props);
if (successConvert)
InsertSchemaPostgres.executeSchemaChange(dbName, props);
break;
case "translate":
// location of the script to allow results from Postgres to be
// piped back to this class. Adapt the scripts if necessary.
String scriptLoc = "C:/Users/ocraw/IdeaProjects/cyp2sql-next/pgdbPlay.bat";

// Cypher query to translate and then execute.
String cypher = "MATCH (a:Meta) RETURN DISTINCT a.type " +
"UNION MATCH (a:Process) RETURN DISTINCT a.type;";

try {
// obtain the intermediate representation
DecodedQuery dQ = C2SMain.getDQ(cypher, props);
System.out.println(dQ.getUnionParts().get(0).getMc());

// convert the intermediate representation to SQL
String sql = C2SMain.getTranslation(cypher, dQ, props);
System.out.println(sql);

// execute directly on Postgres if desired (the script will pipe
// the results back into this tool).
String postgresOutput = C2SMain.runPostgres(sql, dbName, scriptLoc);
System.out.println(postgresOutput);
} catch (IOException e) {
e.printStackTrace();
}
break;
default:
System.err.println("Not a valid option...");
System.exit(1);
}
}
}
```

The library and its dependencies can be added to a local Maven repository in the following way:

```
$ git clone git@gitlab.dtg.cl.cam.ac.uk:fresco-projects/cyp2sql-next.git
$ cd cyp2sql-next
$ mvn install
```

2 Getting started

The code for the tool can be found in the *cyp2sql-next* repository: https://gitlab.dtg.cl.cam.ac.uk/fresco-projects/cyp2sql-next/.

There is an initial setup procedure involved before the tool can be run. View the README.md for additional information.

NOTE: Javadoc documentation is also available through the repo.

NOTE: a .JAR file is included within the repo that can be run from the command line.

2.1 Prerequisites

- Java 1.8 or above
- Neo4j, any version should suffice
- **An existing graph database** in Neo4j
- A target database engine for the queries to be translated for, such as Postgres
- Windows or Unix OS
- For large databases, a relatively powerful machine and available disk space (additionally, as much RAM as possible)

2.2 File system preparation

The easiest way to use the tool is to create a new folder, and to also create/add the following folders/files:

- A folder for workspace usage
  - Metafiles created during the schema conversion will be stored here
- A blank text file for storing the keys of the graph that may contain lists – see §4.2.1
- A folder for storing the results from both the Neo4j database, and the target database. For example, create a folder called `results`, and then two files within this folder called `results\neo4jResults.txt` and `results\postgresResults.txt`
- A new Postgres database should also be created – in Unix this can be done via the command `createdb <nameOfNewDatabase>`

2.3 Generating dump files from Neo4j

The tool builds up an equivalent schema in the target database by parsing a text file that is generated via the Neo4j console.

NOTE: the console/Java application offered by Neo4j is currently being deprecated in favour of the new ‘cypher-shell’.

NOTE: the dump file may be very large for large graph databases.

2.3.1 Windows OS

```
java -classpath "C:\Program Files\Neo4j CE 3.0.6\bin\neo4j-desktop-3.2.2.jar"
    ↪ org.neo4j.shell.StartClient -path "C:\Users\ocraw\Documents\CL
    ↪ Internship\Neo4j graphs\Test1KOPUS" -c dump > "C:\Users\ocraw\Documents\CL
    ↪ Internship\Graph Dumps\dumpOPUS1k.txt"
```

2.3.2 Unix OS

```
neo4j-shell -path ../data/databases/dump_prov1000.graphdb/ -c dump >
    ↪ 2017-08-01-dump.txt
```

2.3.3 Dump file explanation

The dump file should look something like:

```
begin
commit
begin
create (_0:`Local` {`mono_time`:"10", `name`:"omega", `node_id`:1, `ref_count`:1, ...
create (_1:`Global` {`name`:["nginx"], `node_id`:2, `sys_time`:-782642514, `type`:2})
create (_2:`Global` {`name`:["nginx"], `node_id`:3, `sys_time`:-782642514, `type`:2})
```

The tool will remove the unnecessary lines from the file, leaving two distinct types remaining to parse into a relational schema: nodes and edges (relationships).

2.4 Configuring the properties

The file `c2s_props.properties` contains all the necessary parameters to allow the tool to function correctly. This file should be edited before running the tool.

neo4jSchema	Path of the dump file generated from the existing Neo4j graph.
workspaceLocation	Path to workspace used by the tool.
neo4jResults	Path for the results from the Neo4j database to be stored into, when a Cypher query is executed.
sqlResults	Path for the results from the relational database to be stored into, when an SQL query is executed.
listsLocation	Path to the text file containing the list of fields that may contain lists.
neo4jUser	Neo4j graph database username.
neoPW	Neo4j graph database password.
postgresUser	Postgres database username.
postgresPW	Postgres database password.

3 Schema conversion

The first step of the pipeline in translating Cypher to SQL is to convert the Neo4j graph database to a relational database. Using the dump file generated from the Neo4j graph, multiple relations can be built up and executed on a relational database backend. During this process, other metafiles are also created – these are used throughout the translation process later for completeness, and in some cases, optimisations.

The tool can currently handle the following datatypes: `TEXT`, `INT`, `REAL`, `BOOLEAN`, and `TEXT[]`.

3.1 Converting the graph database to a relational schema

For the tool to run successfully and without error, make sure all the properties in the properties file are correct. Next, a blank relational database needs to be created. The tool currently will not overwrite existing databases – if an error has occurred when writing to the database, it must be wiped clean before the tool can run again.

```
-s C:\Users\ocraw\IdeaProjects\Cypher_SQL_Translation\c2s_props.properties testDB
```

- The first argument (`-s`) signals to the program that it should be performing the schema conversion
- The second argument is the location of the properties file
- The third argument is the name of the recently created blank database

If the tool runs successfully and without error, the following should have happened:

- The recently created database will now be populated with tables and data – see §3.2
- The workspace folder will now contain four metafiles – see §3.3

NOTE: deleting the workspace folder and its metafiles will cause some translations to fail. If this is done accidentally, the whole graph schema must be converted again from scratch.

3.2 Graph to relational: theory

All the nodes from the graph are stored in one relation (*Nodes*), and all the relationships are stored in a separate relation (*Edges*).

There are some optimisations and additional implementation aspects. Storing all 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 also has its own relation. The translator tool decides at run time the optimal relation to join.

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

For graph functions to be computable on the relational side, the schema converter also builds two materialised views - `adjlist_from` & `adjlist_to`. For the performance of the queries to be better when relationships are used, a unique index constraint is added to the *Edges* relation as well. It is recommended to add even further indices where applicable for larger boosts in speed.

Some other functions are also committed to the database at this point:

`array_unique(arr anyarray)` Helper function for the `cypher_iterate` function (see below). This function takes an array of any type as input, and returns an array of the same type, but with duplicate elements removed.

`cypher_iterate(integer[])` Function for performing the iterate function designed for this project.

`doforeachfunc(integer[], field TEXT, newv TEXT)` Function to help perform the semantics of Cyphers `FOREACH` keyword.

NOTE: the class `PostgresConstants.java` contains the list of additional materialised views and functions committed to the database during the schema conversion process.

Figure 2 shows in detail an example conversion (the example is based on an OPUS graph with 100,000 nodes).

3.3 Metafiles created during schema conversion

`meta_nodeProps.txt` – **Creation:** all properties of the nodes are stored in this file. **Usage:** currently two major uses. One is to help the Neo4j database driver to return the correct results to the user (for example, when a wildcard is used in the query). Secondly, it is used in the cases where a GROUP BY is required in the SQL statement to be executed.

`meta_labelProps.txt` – **Creation:** a slightly more detailed version of the file above. The file contains the list of properties of all the nodes, but is sectioned into the individual label types of the nodes.

Example:

```
*country*
name
population
dialling_code
*city*
name
population
state
ave_house_price
```

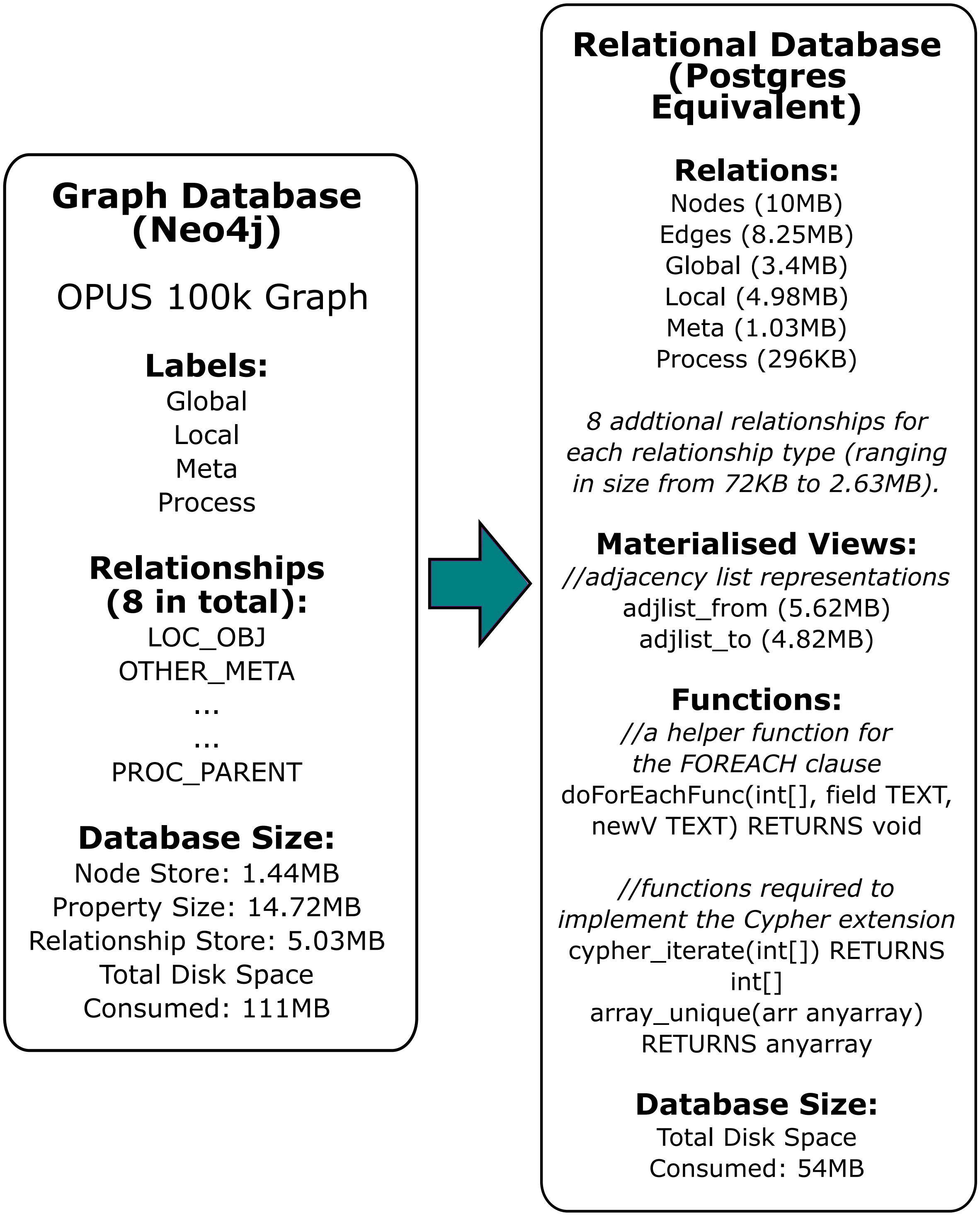
Usage: the file is used to see if any possible optimisations to the SQL query are achievable. This is done by finding unique nodes for a property. In the example above, if the property/key 'state' only belongs to the label 'city', then the tool can use this information to its advantage.

See the method `getLabelMapping()` in `C2SMain.java` for further information. Information from this file is stored in the `labelProps` map (publicly accessible using `C2SMain.labelProps`).

`meta_rels.txt` – **Creation:** all the types of edges in the Neo4j graph are stored in this file. **Usage:** currently only used in the case where the tool can be used to delete information from the database (this feature has not been thoroughly tested yet though, so caution is advised if a query which is being used to delete data is to be executed).

See the method `getLabelMapping()` in `C2SMain.java` for further information. Information from this file is stored in the `allRelTypes` map (publicly accessible using `C2SMain.allRelTypes`).

`meta_labelNames.txt` – **Creation:** all the names of the labels in the Neo4j graph are stored. **Usage:** another optimisation technique – helps the tool choose the best relation to use to complete the query correctly.



Graph Database (Neo4j)

OPUS 100k Graph

Labels:
Global
Local
Meta
Process

Relationships (8 in total):
LOC_OBJ
OTHER_META
...
...
PROC_PARENT

Database Size:
Node Store: 1.44MB
Property Size: 14.72MB
Relationship Store: 5.03MB
Total Disk Space Consumed: 111MB

Relational Database (Postgres Equivalent)

Relations:
Nodes (10MB)
Edges (8.25MB)
Global (3.4MB)
Local (4.98MB)
Meta (1.03MB)
Process (296KB)

8 addtional relationships for each relationship type (ranging in size from 72KB to 2.63MB).

Materialised Views:
//adjacency list representations
adjlist_from (5.62MB)
adjlist_to (4.82MB)

Functions:
//a helper function for the FOREACH clause
doForEachFunc(int[], field TEXT, newV TEXT) RETURNS void

//functions required to implement the Cypher extension
cypher_iterate(int[]) RETURNS int[]
array_unique(arr anyarray) RETURNS anyarray

Database Size:
Total Disk Space Consumed: 54MB

Figure 2: Schema conversion outline.

4 Translation of queries

4.1 Usage

The tool requires a running Neo4j instance to be up and running.

```
-t \..\..\c2s_props.properties testDB <-dp|-dn|-r>
```

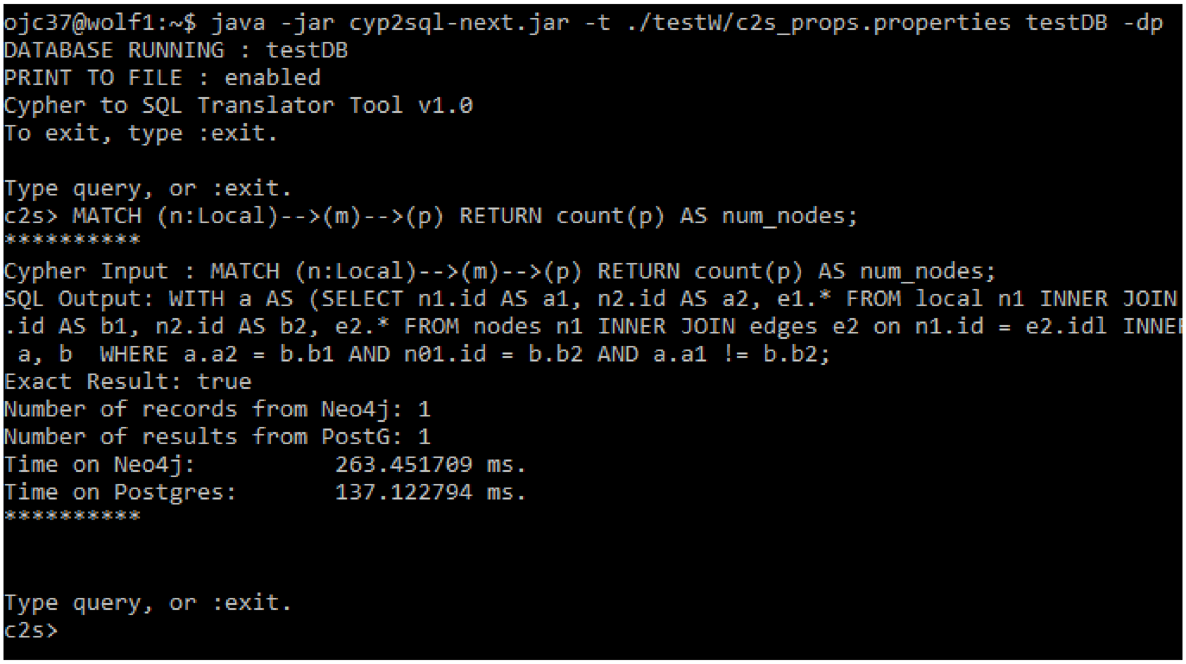
- The first argument (`-t`) signals to the program that it should be performing the translation of queries
- The second argument is the location of the properties file
- The third argument is the name of the relational database that the queries are to be executed on
- The fourth argument is another flag that gives the user some choice in what they want to do – see the table below for further guidance

`-dp or -dn` **Debug interface**. This function allows the user to input queries through the command line interface for translation. Useful for debugging. If `-dp` is used, the results from both the databases are printed to local files. If `-dn` is set, this does not occur.

`-r` **Run as normal**. This function allows the user at the command line to input queries for translation. Unlike `-dp` or `-dn`, this will not execute Cypher on Neo4j, and will only output the results from the target database engine.

NOTE: the script to run Postgres on Unix systems when the tool is in `-r` mode may not have the right permissions to run with the tool. `chmod +x auto_run_pg.sh` should fix the issue.

The user at this point should be presented with a command line interface where Cypher can be typed into:

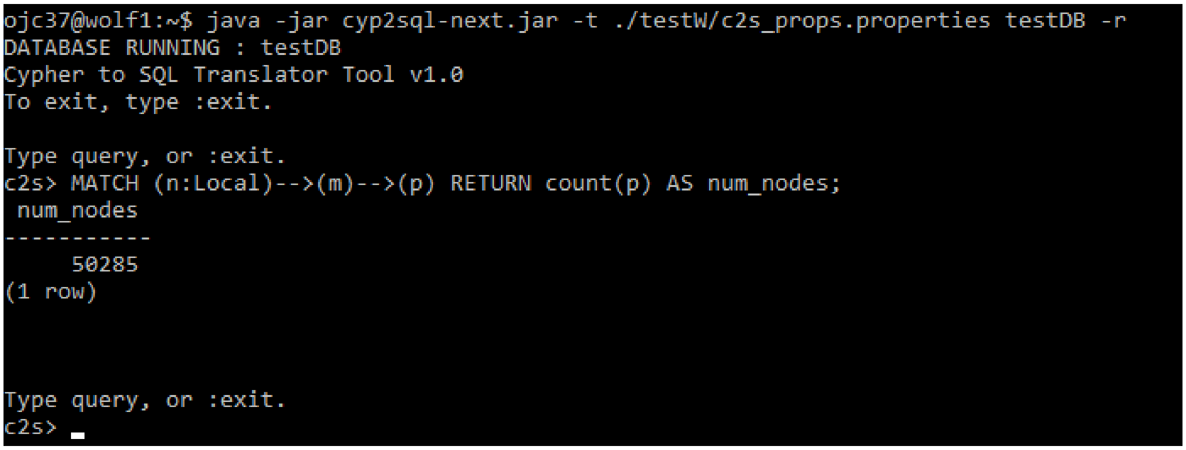


```
ojc37@wolf1:~$ java -jar cyp2sql-next.jar -t ./testW/c2s_props.properties testDB -dp
DATABASE RUNNING : testDB
PRINT TO FILE : enabled
Cypher to SQL Translator Tool v1.0
To exit, type :exit.

Type query, or :exit.
c2s> MATCH (n:Local)-->(m)-->(p) RETURN count(p) AS num_nodes;
**********
Cypher Input : MATCH (n:Local)-->(m)-->(p) RETURN count(p) AS num_nodes;
SQL Output: WITH a AS (SELECT n1.id AS a1, n2.id AS a2, e1.* FROM local n1 INNER JOIN
.id AS b1, n2.id AS b2, e2.* FROM nodes n1 INNER JOIN edges e2 on n1.id = e2.idl INNE
 a, b  WHERE a.a2 = b.b1 AND n01.id = b.b2 AND a.a1 != b.b2;
Exact Result: true
Number of records from Neo4j: 1
Number of results from PostG: 1
Time on Neo4j:          263.451709 ms.
Time on Postgres:       137.122794 ms.
**********


Type query, or :exit.
c2s>
```

Figure 3: Screenshot of debug mode.



```
ojc37@wolf1:~$ java -jar cyp2sql-next.jar -t ./testW/c2s_props.properties testDB -r
DATABASE RUNNING : testDB
Cypher to SQL Translator Tool v1.0
To exit, type :exit.

Type query, or :exit.
c2s> MATCH (n:Local)-->(m)-->(p) RETURN count(p) AS num_nodes;
 num_nodes
-----------
     50285
(1 row)



Type query, or :exit.
c2s>
```

Figure 4: Screenshot of run mode.

4.2 Query translation

The query is first tokenised using the ANTLR generated classes for the Cypher language. See §5 for more details.

An intermediate representation in the form of a DecodedQuery object is the built. This object is then referred to when translating to the target language (such as SQL). See §4.3 for more detailed descriptions.

Multiple different classes can be used to perform the translation to SQL. Likewise, if a new type of translation is needed, the `AbstractConversion.java` and `AbstractTranslation.java` classes can be extended to suit.

4.2.1 Dealing with Cypher lists

Lists in Cypher present one current challenge for the translation tool. Although the translation can be done, there is no current mechanism for detecting cleanly which keys of nodes/relationships may have type list.

To solve this problem (and to possibly fix any issues encountered with lists), a blank file should be created (with a name such as `lists.txt`). This file should then be referred to in the `c2s_props.properties` file (see §2.4).

The file should contain a list, separated by new lines, of each possible key in the graph that may contain a list. This will aid the translator in producing the correct result.

4.2.2 Where clauses

Where clauses are converted into properties of nodes/relationships. For example, if there is a Cypher clause, `MATCH (n) WHERE n.a = 1 AND n.b = 2 RETURN n`, then the translator tool associates the two predicates to the node `n`. When the translation to SQL occurs, the properties of node `n` are then extracted and converted.

The whole where clause is decoded in `CypherTranslator.java`. The initial clause is separated into its individual parts (they will be separated by either AND/OR). A `CypWhere` object is then created for each component, to store:

- The position it appears in the clause (the indexing counting left to right)
- Any brackets that appear before or after the clause (to ensure consistent ordering later)
- The boolean operator that immediately follows it

Depending on what the clause itself is conveying, it is parsed accordingly. When it has been parsed, the properties and semantics need to be stored. As mentioned above, they are stored with the nodes/relationships that they belong with. For example, if the where clause says something like `a.name = 'me'`, the properties and semantics of the clause will be stored with the node that has the id `'a'` (assuming `'a'` is of course a node and not a relationship).

The format of how it is stored with the nodes/relationships is initially quite confusing (and in all honesty a bit silly, could do with a refactor), but can be quickly described with this guide:

```
MATCH p=shortestPath((f {name:"omega"})-[*1..6]->(t:Meta))
RETURN count(t);
```

Property stored with node: `[name="omega"]`

In this example, as the property is described within the node itself, the property is simple to understand and parse.

```
MATCH (a:Meta)
WHERE a.sys_time < 0
RETURN count(a);
```

Property stored with node: `[sys_ time="p1null@null$lt#0#tl"]`

- Already, the syntax looks very different, but can be broken down easily
- `p1` denotes that the component `a.sys_time < 0` is in the first position of the clause (in fact, in this example, it is the only component)
- The `null` before the @ symbol means there is no boolean operator (such as AND or OR) that follows this component
- The `null` after the @ symbol means that the component is not associated with any bracketing
- The value after the $ sign, `lt#0#tl` describes the predicate itself:
  - `lt` is the code for the < symbol
  - The operand 0 is contained within the # symbols

```
MATCH (a)
WHERE a.node_id < 345
OR (
(a.node_id > 800 AND 'Process' in labels(a))
OR a.node_id = 983
)
RETURN count(a);
```

Property stored with node: `[node_id="p1or@null$lt#345#tl~p2and@(($gt#800#tg~p4null@)$eq#983#qe", label="p3or@)$eq#process#qe"]`

- In this case, two fields of the node have properties associated with them
- When multiple properties apply to one field, the ~ symbol separates the individual components that will be translated
- When brackets are involved (notice in the original Cypher input the 2 left brackets before `a.node_id > 800`), these are stored after the @ symbol (in their literal form)
  - Extra care must be taken in the code in the case of functions such as `labels(a)` at the end of a component (only one bracket should be recorded and not two)

`eq#...#qe`	Equality: `WHERE idA.property = someValue`.
`ne#...#en`	Not equal to: `WHERE idA.property <> someValue`.
`lt#...#tl`	Less than: `WHERE idA.property < someValue`.
`gt#...#tg`	Greater than: `WHERE idA.property > someValue`.
`le#...#el`	Less than or equal to: `WHERE idA.property <= someValue`.
`ge#...#eg`	Greater than or equal to: `WHERE idA.property >= someValue`.
`ex#...#xe`	Exists function: `WHERE exists(idA.property)`.
`nx#...#xn`	Exists function (not): `WHERE not exists(idA.property = someValue)`.
`in#...#ni`	IN: `WHERE ‘someLabel’ in labels(idA)`.
`anyin#...#niyna`	Any (where IN is also used): `WHERE any(var IN idA.property WHERE var IN [someList]`.
`anyeq#...#qeyna`	Any (where IN is also used): `WHERE any(var IN idA.property WHERE var = someValue`.
`isn#...#nsi`	IS NULL: `WHERE idA.proeprty IS NULL`.
`non#...#non`	IS NOT NULL: `WHERE idA.proeprty IS NOT NULL`.

4.2.3 Iterate semantics

As part of an extension to my original dissertation on this tool, additional functionality to the graph model was included. 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 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 below:

```
ITERATE MATCH (a {host: ‘google’})-->(b:Website)
-->(c {domain: ‘net’})
LOOP c ON b COLLECT n
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 the 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 5.

Figure 5 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:

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

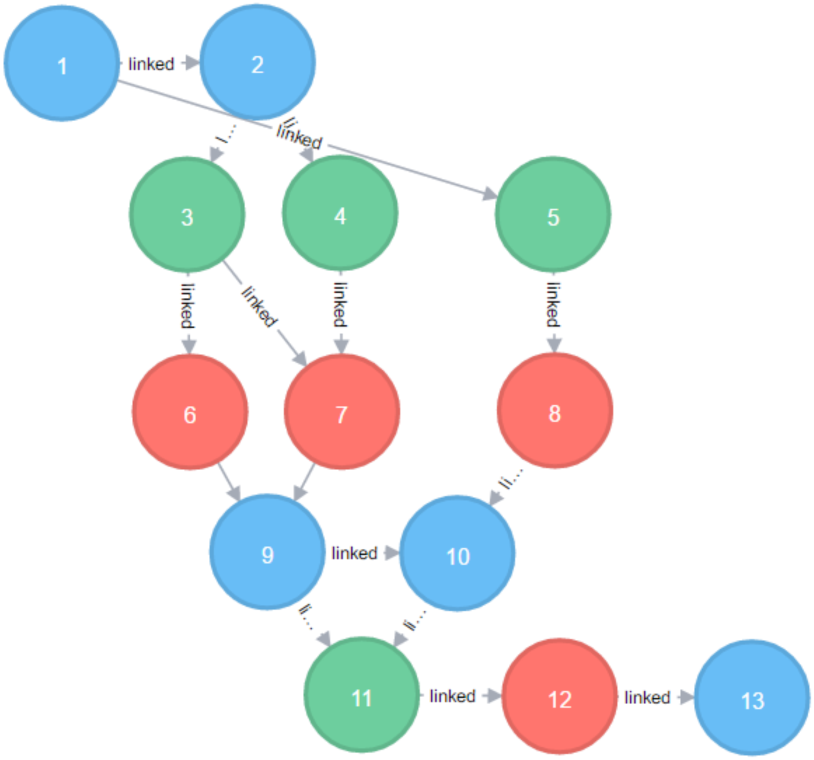


Figure 5: A trivial graph consisting of thirteen nodes with three different labels – blue for ‘l1’, green for ‘l2’, and red for ‘l3’.

Executing this query on Postgres returns the results shown in Table 1.

Table 1: Results of Iterate Query.

n.uid
9
9
9
10
13
13
13
13

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.

4.3 Workflow through the code

On the following pages is an example of how a query is translated from Cypher to SQL through the tool. Included with the flow are descriptions to help the reader.

Cypher input:

```
MATCH (n:Process)<-[e:PROC_OBJ]-(c:Local)
WHERE id(n) = 916 AND e.state in [5]
RETURN c.name, e.state
ORDER BY c.name DESC;
```

The Cypher should ideally be inputted with the correct style, such as capitalised keywords and spaces in the appropriate places.

Filter on keywords to appropriate parsing class:

FOREACH = *ForEach_Cypher.java*

WITH (more than one use of the keyword) = *Multiple_With_Cypher.java*

WITH (just one occurrence) = *With_Cypher.java*

shortestPath = *SP_Cypher.java*

ITERATE[1] = *Iterate_Cypher.java*

The first step is to determine the best class for which to handle the initial Cypher input, based on the keywords it contains. This happens in the *translateCypherToSQL()* method in *C2SMain.java*. The tool filters the Cypher based on the options (in order) shown on the left.

If the input goes through these filters, then it is passed to the *AbstractConversion* class. This class is the **superclass** of all the classes mentioned.

The *AbstractConversion* class contains the method *convertCypherToSQL()*, which in turn performs some additional parsing on the Cypher input based on the *UNION*/*UNION ALL* keyword.

Tokenisation:

```
[match, (, n, :, process, ), <, -, [, e, :,
proc_obj, ], -, (, c, :, local, ), where, id, (,
n, ), =, 916, and, e, ., state, in, [, 5, ],
return, c, ., name, ,, e, ., state, order, by,
c, ., name, desc]
```

The Cypher is parsed by the ANTLR framework in the method *getTokenList()* – this produces a list of tokens (excluding spaces, EOF, ;, and any return aliases), and it also creates a *CypherWalker()* object.

Cypher Walker object:

```
hasDistinct = false         hasCount = false
hasCollect = false          hasCase = false
hasDelete = false

matchClause =
   "(n:Process)<-[e:PROC_OBJ]-(c:Local)"
returnClause = "c.name, e.state"
returnAlias = {HashMap@2938}  size = 0
orderClause = "c.name desc"
latestOrderDirection = "desc"
skipAmount = -1
limitAmount = -1
```

A *CypherWalker* object, which extends one of the automatically generated ANTLR classes, uses the parse tree to obtain additional information to the tokens.

[1] *ITERATE* is not an accepted keyword in Cypher currently (and therefore will throw an error on Neo4j), but the semantics of the keyword can be translated to SQL (see **Error! Reference source not found.** for more information.)

DecodedQuery object:

MatchClause object (matchC):

```
nodes = {ArrayList@2994}  size = 2
 0 = {CypNode@2998} "(ID: n, LABELS:
process, PROPS: null, POS: 1)"
 1 = {CypNode@2999} "(ID: c, LABELS: local,
PROPS: null, POS: 2)"

rels = {ArrayList@2995}  size = 1
 0 = {CypRel@3003} "(ID: e, TYPE: proc_obj,
DIR: left, PROPS: null, POS: 1)"

varRel = false
internalID = 1
```

ReturnClause object (returnC):

```
items = {ArrayList@3012}  size = 2
 0 = {CypReturn@3070} "(ID: c, FIELD: name,
TYPE: node, POS: 2, COUNT: false, COLLECT:
false, CASE: null)"
 1 = {CypReturn@3071} "(ID: e, FIELD: state,
TYPE: rel, POS: 1, COUNT: false, COLLECT:
false, CASE: null)"
```

OrderClause object (orderC):

```
items = {ArrayList@3077}  size = 1
 0 = {CypOrder@3108} "(ID: c, FIELD: name,
ORDER_TYPE: desc)"
```

Translation to SQL filter:

NoRels.java = no relationships in the match clause. For example, MATCH (n) RETURN n

*SingleVarAdjList.java = if a variable length path is used (and specifically, only between two nodes). For example, MATCH (a)-[*1..3]->(b) RETURN b*

The next step in the workflow is to generate a *DecodedQuery* object: this is the object that encompasses all the intermediate representation calculated by the tool. This is achieved through the *generateDecodedQuery()* method of *CypherTranslator.java*.

The MATCH part of the Cypher input is firstly decoded. The information gathered is stored in a *MatchClause* object. The nodes are themselves objects of the type *CypNode*; the relationships are stored in *CypRel* objects.

varRel indicates if the relationship in the MATCH part of the Cypher input is of the type –[*a…b]-

The RETURN part is next to be decoded. The tool handles the individual elements of the return clause by splitting the whole clause by “,”.

Each individual component is then parsed and converted into a *CypReturn* object, which itself is stored in a *ReturnClause* object.

The ORDER BY part of the Cypher input is then decoded (in a similar style as the RETURN clause), and stored in an *OrderClause* object.

If the Cypher input contains SKIP or LIMIT values, these are also recorded and stored in the *DecodedQuery* object. The CypherWalker object mentioned earlier is also stored in the *DecodedQuery* object.

The resulting *DecodedQuery* object is now passed as an argument to the method *SQLTranslate.translateRead*() method (the return type of this method is the SQL string; this is then stored in the *DecodedQuery* object).

MultipleRel.java = if one or more relationships are present in the match clause. For example, MATCH (a)--()--()-->(b) RETURN b

There are three different translation types for the tool to currently decide between – these are shown on the left.
In this example, the *MultipleRel* class will be chosen. All 3 of these classes extend the base class *AbstractTranslation()*.

Generation of SQL:

```
WITH a AS (SELECT n1.id AS a1, n2.id AS a2,
e1.* FROM process n1 INNER JOIN e$proc_obj
e1 on n1.id = e1.idr INNER JOIN local n2 on
e1.idl = n2.id WHERE ( n1.id = '916'  ) AND
e1.state IN (5) )
```

```
WITH a AS (SELECT n1.id AS a1, n2.id AS a2,
e1.* FROM process n1 INNER JOIN e$proc_obj
e1 on n1.id = e1.idr INNER JOIN local n2 on
e1.idl = n2.id WHERE ( n1.id = '916'  ) AND
e1.state IN (5) ) SELECT n01.name, a.state
FROM nodes n01, a
```

```
WITH a AS (SELECT n1.id AS a1, n2.id AS a2,
e1.* FROM process n1 INNER JOIN e$proc_obj
e1 on n1.id = e1.idr INNER JOIN local n2 on
e1.idl = n2.id WHERE ( n1.id = '916'  ) AND
e1.state IN (5) ) SELECT n01.name, a.state
FROM nodes n01, a  WHERE n01.id = a.a2
```

```
WITH a AS (SELECT n1.id AS a1, n2.id AS a2,
e1.* FROM process n1 INNER JOIN e$proc_obj
e1 on n1.id = e1.idr INNER JOIN local n2 on
e1.idl = n2.id WHERE ( n1.id = '916'  ) AND
e1.state IN (5) ) SELECT n01.name, a.state
FROM nodes n01, a  WHERE n01.id = a.a2
ORDER BY n01.name desc
```

Within the *translate*() method of *MultipleRel*, the SQL is built up as follows from the *DecodedQuery* object:

- The CTEs (WITH parts of SQL) are individually composed for each relationship in the original match clause of Cypher
- The SELECT .. FROM part of SQL is then generated and appended to the previous CTEs
- Any predicates that should be in the WHERE part of SQL is then also appended
- Any ORDER BY components are then added

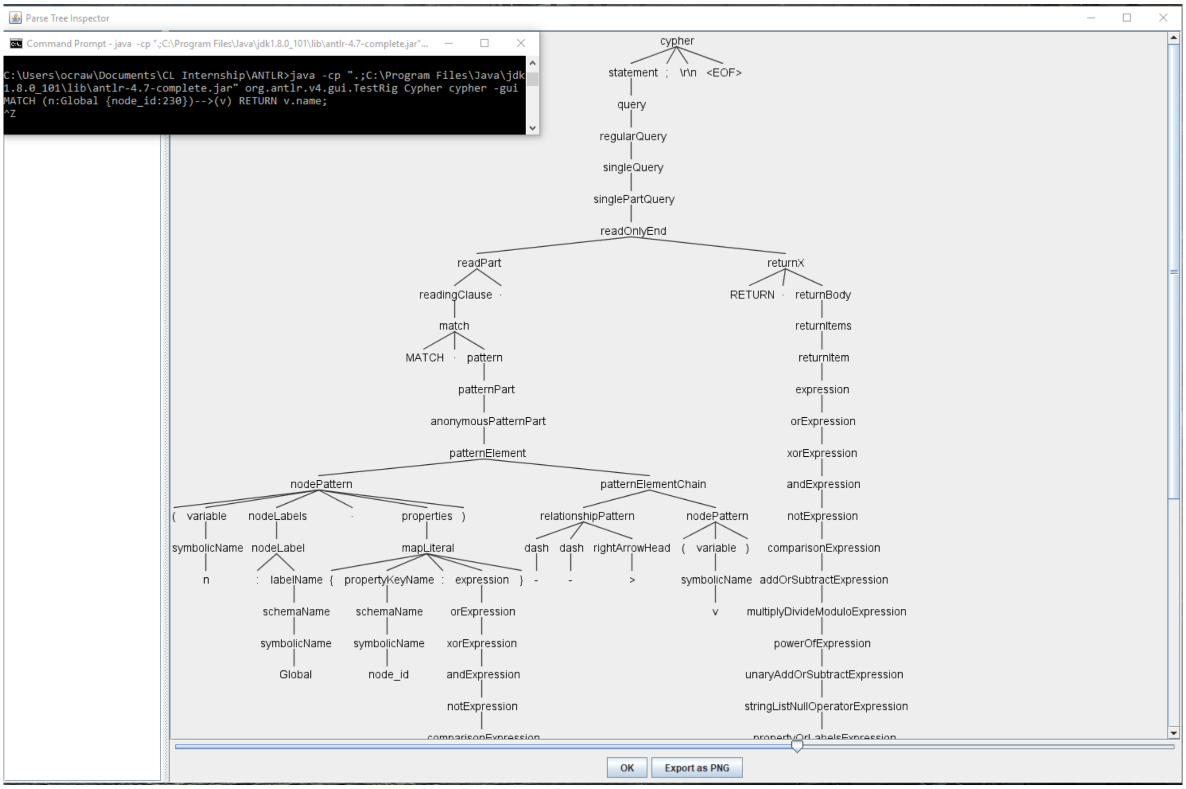
The resulting SQL is then executed.

5 ANTLR framework

The current tool uses ANTLR v4[1] as a way of parsing the initial Cypher query, both into tokens and as a more effective way of parsing information out of the query. The ANTLR framework requires a grammar of Cypher to work, and this has been obtained from the *openCypher* project[2].

The guide for setting up the parsers and lexers has been adapted from the ANTLR v4 'Getting Started' guide[3].

NOTE: the parser and lexer should already be working and configured within the *Cyp2SQL* code itself. The information below would be useful in the case of an update to either the Cypher grammar file, or to the ANTLR library. See the package `parsing_lexing` for the existing configuration.



```
C:\Users\ocraw\Documents\CL Internship\ANTLR>java -cp ".;C:\Program Files\Java\jdk
1.8.0_101\lib\antlr-4.7-complete.jar" org.antlr.v4.gui.TestRig Cypher cypher -gui
MATCH (n:Global {node_id:230})-->(v) RETURN v.name;
^Z
```

Figure 6: Parse tree of a Cypher query with ANTLR.

5.1 ANTLR setting up

Firstly, a new temporary folder should be made – add to it the latest `Cypher.g4` grammar file.[4] Then run the following command:

```
java -jar "C:\Program Files\Java\jdk1.8.0_101\lib\antlr-4.7-complete.jar" Cypher.g4
```

Followed by this command:

```
javac -classpath .;"C:\Program Files\Java\jdk1.8.0_101\lib\antlr-4.7-complete.jar"
    ↪ Cypher*.java
```

[1] http://www.antlr.org/download.html

[2] https://s3.amazonaws.com/artifacts.opencypher.org/M06/Cypher.g4

[3] https://github.com/antlr/antlr4/blob/master/doc/getting-started.md

[4] One of the keywords in the original file was named `return`, but this clashes with the tool, and so it has been renamed to `returnX`

It is a good idea to test the installation at this point:

```
java -cp ".;C:\Program Files\Java\jdk1.8.0_101\lib\antlr-4.7-complete.jar"
    ↪ org.antlr.v4.gui.TestRig Cypher cypher -gui
```

Assuming a valid installation, commands can now be typed into the console - `^Z` will end the input and hopefully produce an image, such as the one in Figure 6.

The source code files generated by the tool are contained within the package `parsing_lexing`.

6 Current Cypher translation list

The tool cannot currently translate the complete set of Cypher available to use. Below is a detailed guide of what can be translated. Further translations are more than possible by viewing/adapting the source code to suit.

NOTE: information gathered below has been adapted from the full list of Cypher commands obtained from the Neo4j Cypher reference card: https://neo4j.com/docs/cypher-refcard/current/

NOTE: quantifiers from regular expressions may be used below to make the list more concise. As a quick reference, * = zero or more times; + = one or more times; ? = zero or one time.

MATCH ... RETURN ...

The queries below will be translated via the `NoRels.java` class.

```
MATCH (idA)
RETURN [DISTINCT]? idA

MATCH (idA:labelA)
RETURN idA.propertyA

MATCH (idA:labelA:labelB {keyA:valueA})
RETURN idA.propertyA AS alias_value, idA.propertyB AS other_alias

MATCH (idA)
RETURN [count(idA)|collect(idA.propertyA)|sum(idA.propertyA)|
min(idA.propertyA)|max(idA.propertyA)|avg(idA.propertyA)]
```

ORDER BY, SKIP, and LIMIT

Queries containing these keywords need only a small adjustment to translate.

```
MATCH (idA)
RETURN idA.propertyA
ORDER BY idA.propertyB [ASC|DESC]?
 [SKIP skipValue]? [LIMIT limitValue]?
```

MATCH ... WHERE ... RETURN

These are also translated to SQL in the class NoRels.java.

```
MATCH (idA)
WHERE [NOT]? idA.propertyA [=|<>|<|>|<=|>=] value
RETURN count(idA)

MATCH (idA)
WHERE idA.propertyA < valueA [AND|OR] idB.propertyB > valueB
RETURN count(idA)
```

Paths – MATCH ()-->()-->() ...

These are translated to SQL in the class MultipleRel.java.

```
MATCH (idA)-->(idB)-->(idC)-->(idD)
WHERE id(idD) < some_value
RETURN count(idA) AS some_alias

MATCH (idA)<-[[relID]?:REL_TYPE]-(idB)
RETURN idB
ORDER BY idB.propertyA ASC

MATCH ()-[r:REL_TYPE {keyA:valueA}]-()
RETURN r.propertyA
```

With – ... WITH ... RETURN

These are translated to SQL in the class WithSQL.java, although additional parsing of the Cypher input will also occur in either the class With_Cypher.java or Multiple_With_Cypher.java.

```
MATCH (idA:labelA)-[rel]->(idB:labelB)
WITH idA, count(rel) AS alias
WHERE alias > value
RETURN idA.propertyA, alias
ORDER BY alias DESC

MATCH (idA)
WITH idA
ORDER BY idA.propertyA DESC
LIMIT 3
RETURN collect(idA.propertyA)

MATCH (idA)
WHERE someLabel IN labels(idA)
WITH idA
MATCH (idB)
WHERE idB.propertyB = idA.propertyA
RETURN count(idA)
```

UNION [ALL]?

The individual components of the input are dealt with normally, and then combined together with the correct UNION semantics in the class UnionSQL.java.

```
MATCH (idA)-[:rel1]->(idB)
RETURN idB.propertyA
[UNION|UNION ALL]
MATCH (idA)-[:rel2]->(idB)
RETURN idB.propertyA
```

Variable length paths – -[*x..y]->

This subset of Cypher is dealt with using the SingleVar.java class.

```
MATCH (idA:labelA)-[*x..y]->(idB:labelB)
RETURN count(idB)

//x and y represent integer values for the length of the path
//being traversed. These queries, unlike the paths shown earlier,
//only currently work for one pattern - it cannot be extended
//in the following way:

MATCH (idA:labelA)-[*x..y]->(idB:labelB)-[*x2..y2]->(idC)
RETURN count(idC)

//furthermore, a direction must be specified.
```

Shortest path queries

This subset of Cypher is dealt with using the ShortestPath.java class.

```
MATCH p=shortestPath((idA {key:value})-[*1..x]->(idB:labelB))
RETURN count(idB)
```

Functions – exists(), labels(), id()

These keywords are parsed in the class CypherTranslator.java, where they are stored with the node/relationship that they are referring to.

```
MATCH (idA)
WHERE exists(idA.propertyA) AND exists(idA.propertyB)
RETURN count(idA)

MATCH (idA)-[rel]->(idB)
WHERE NOT id(idB) = value OR "someLabel" IN labels(idB)
RETURN idA.propertyA
```

... WHERE ... IN and any()

These keywords are parsed in the class `CypherTranslator.java`, where they are stored with the node/relationship that they are referring to.

```
MATCH (idA)
WHERE id(idA) IN [val1, val2, ..., valN]
RETURN idA.propertyA

MATCH (idA)
WHERE any(someV IN labels(idA) WHERE someV IN [val1, ..., valN])
RETURN count(idA)

MATCH (idA)
WHERE any(someVar IN idA.propertyA WHERE someVar = value)
RETURN count(idA)
```

ITERATE MATCH ...

This additional keyword is parsed and translated in the class `Iterate_Cypher.java`. See §4.2.3 for more details.

```
ITERATE MATCH (idA)-->(idB)-...->(idX)
LOOP idX ON idA COLLECT idN
RETURN idN.propertyA
```

7 Known issues

- The use of `WITH` is still quite constrained in its use
- Alias names cannot be the same as field names – for example, `a.points AS points` would not work; `a.points AS points_x` is fine however
- Some labels from Neo4j may map to reserved keywords in SQL (such as *Group* and *User*) - in these cases the optimised table for these labels cannot be created
- When predicates in the `WHERE` clause are added, they may sometimes lead to incorrect translations (mostly in the `WITH` parts of a multiple relationship Cypher input). It can be solved in some cases by adding the predicates in a different way in the original input (see example below)

```
MATCH (a)-[e]->(b)-[f]->(c)
WHERE a.type = b.type AND c.pid < b.pid
RETURN count(f);

// instead of

MATCH (a)-[e]->(b)-[f]->(c)
WHERE a.type = b.type AND b.pid > c.pid
RETURN count(f);

// where 'b' now has 2 properties associated with it,
// and will not work (currently).
```

8 List of test queries from OPUS

```
MATCH p=shortestPath((f {name:"omega"})-[*1..6]->(t:Meta)) RETURN count(t);

MATCH (a)-[*1..3]->(c:Process) RETURN count(c);

MATCH (a:Local)-[*4..9]->(b)
RETURN DISTINCT b.node_id, b.sys_time AS time_alias ORDER BY b.node_id DESC;

MATCH (n {name:["/var/db/entropy/saved-entropy.7", "/var/db/entropy/saved-entropy.8"]})
RETURN n.node_id ORDER BY n.node_id ASC;

MATCH (a:Global)-[]->(b) RETURN b.node_id AS conn_id;

MATCH (a:Global)-[]->(b) RETURN count(b.node_id);

MATCH (a:Global)-->(b:Local)-->(c:Process)<--(d:Local)<--(b) RETURN count(b);

MATCH ()-[r]-() RETURN DISTINCT r.state ORDER BY r.state;

MATCH (n:Local)<--(m:Global)
RETURN m.node_id AS thing, m.type AS ty ORDER BY m.sys_time LIMIT 3;

MATCH (a:Global) RETURN count(a) AS funky;

MATCH (a:Meta) WHERE a.sys_time < 0 RETURN count(a);

MATCH (a:Meta) WHERE a.sys_time < 0 OR a.node_id > 845 RETURN count(a);

MATCH (a)-[r:LOC_OBJ]-(b) RETURN b.name, r.state ORDER BY b.node_id ASC LIMIT 15;

MATCH ()<-[r:LOC_OBJ {state:12}]-(idA {type:2}) RETURN count(r);

MATCH (n) WHERE id(n) = 345 RETURN n.mono_time, n.sys_time, n.name;

MATCH (a)-->(b)-->(c)-->(d) WHERE id(d) < 123 RETURN count(a) AS cool;

MATCH (n) WHERE exists(n.value) AND exists(n.timestamp) RETURN count(n);

MATCH (n)--()--()--()--(n) WHERE exists(n.status) RETURN count(n);

MATCH (s)-[e]-(d)
WHERE id(s) = 349 AND NOT 'Process' in labels(s) AND NOT 'Global' in labels(d)
RETURN d.node_id ORDER BY d.node_id ASC;

MATCH (n:Process)<-[e:PROC_OBJ]-(c:Local)
WHERE id(n) = 916 AND e.state in [5]
RETURN c.name, e.state ORDER BY c.name DESC;

MATCH (a)-[e]-(b)
WHERE id(a) IN [100, 200, 300, 400] AND id(b) IN [101, 201, 202, 302, 404]
RETURN e.state;

MATCH (a) WHERE any(name in a.name WHERE name = 'uid') RETURN count(a);

MATCH (a) WHERE any(lab in labels(a) WHERE lab IN ['Global', 'Meta'])
RETURN count(a);

MATCH (n) WHERE 'Process' in labels(n)
WITH n
MATCH (m) WHERE m.status = n.status RETURN count(n);
MATCH (n) WHERE 'Local' in labels(n) AND NOT exists(n.pid)
WITH n
MATCH (m:Global)-[r]->(n) WHERE id(m) > 900 RETURN n.node_id, r.state;

MATCH (n:Global)-->(m:Local) WHERE n.node_id < m.node_id RETURN count(m);

MATCH (n:Meta)<--(m:Process)-->(p)
WHERE n.node_id > m.node_id AND p.node_id <= m.node_id RETURN count(m);

MATCH (n) WHERE id(n) < 3
WITH n
MATCH (m) WHERE id(m) < id(n)
WITH m
MATCH (p) WHERE p.node_id < m.node_id RETURN count(p);

MATCH (n) WHERE 'Meta' in labels(n) OR any(name in n.name WHERE name = 'postgres')
WITH n
MATCH (m:Process) WHERE id(m) > id(n)
WITH m
MATCH (p)-->(m)
WITH p
MATCH (j)<-[:PROC_OBJ_PREV]-(p) WHERE p.sys_time = j.sys_time RETURN count(j);

MATCH (a:Global {name:'postgres'})-->(b:Global)
WITH b
MATCH (c) WHERE c.sys_time = b.sys_time
WITH c
MATCH (c)<--(d) RETURN DISTINCT d.node_id ORDER BY d.node_id LIMIT 5;

MATCH (a:Meta) RETURN count(distinct a.name);

MATCH (a:Local)-->(b)<--(c:Process)<--(d) RETURN min(d.node_id);

MATCH (n)
WHERE 'Global' in labels(n) AND any(name in n.name WHERE name = 'master')
OR (exists(n.pid) AND n.status = 2)
WITH n
MATCH (m:Meta) WHERE m.node_id > n.node_id RETURN DISTINCT n LIMIT 10;

MATCH (a)
WHERE a.node_id < 345
OR ((a.node_id > 800 AND 'Process' in labels(a)) OR a.node_id = 983)
RETURN count(a);

MATCH (a)
WHERE (any(x in a.name where x = 'master')
OR any(y in a.value where y in ['postgres', 'nginx']))
AND ('Global' in labels(a) OR 'Meta' in labels(a))
RETURN count(a);

MATCH (a)-[e]->(b)-[f]->(c)
WHERE a.type = b.type AND c.pid < b.pid RETURN count(f);

MATCH (a)-[z]->(b)-[w]->(c)
WHERE a.node_id < b.node_id RETURN w.state, c.type ORDER BY c.node_id ASC;

MATCH (a)-[e]->(b:Process) WHERE e.state > 5
WITH b
```

```
MATCH (c) WHERE (exists(c.pid) AND c.pid < b.pid)
WITH c
MATCH (c)<--(d:Local)
WHERE any(n in d.name WHERE n = '4') RETURN count(d) AS cool_thing;

MATCH (a)-->(b)-->(c) WHERE c.node_id < b.node_id
WITH c
MATCH (d)--(c) WHERE exists(d.ref_count)
WITH d
MATCH (e)-->(d)<--(f) WHERE f.node_id > e.node_id
WITH f
MATCH (g)<-[ww]-(f) WHERE ww.state = 5
WITH g
MATCH (g)-->(ii)-->(i) RETURN DISTINCT i.node_id ORDER BY i.node_id ASC;

MATCH (a)-->(b)-->(c:Process)
WHERE a.node_id < b.node_id
RETURN a.node_id, b.node_id, case c.type when 3 then 'boo' else 'hiss' end

MATCH (a:Global)-[:LOC_OBJ]->(b)
WITH a, count(b) AS num_things WHERE num_things > 2
RETURN a.name ORDER BY a.sys_time DESC;
```