

Cyp2SQL

Documentation

Version 1.2

Last edited: 11/08/2017 14:57 by Oliver Crawford (ojc37@cam.ac.uk)

Table of Contents

Overview of the tool.....	3
Quick Usage	3
Getting Started.....	4
Prerequisites	4
File System Preparation	4
Generating a dump file from a Neo4j database	4
Windows.....	5
Unix.....	5
Configuring the properties.....	6
Schema Conversion	7
Converting the graph database to a relational schema	7
Graph to relational - theory	7
Metafiles created during schema conversion	10
Translation of Queries.....	11
Usage	11
Query Translation	12
Tokenisation.....	12
Intermediate Representation	13
Translation to SQL.....	13
Workflow through the code.....	18
ANTLR Framework	22
Current Cypher translation list	24
Appendices.....	30
Appendix A - overview of translation of a Cypher query to SQL.....	30

Appendix B - current list of test queries from OPUS	31
---	----

Overview of the tool



Figure 1 - Complete pipeline of tool.

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

- A *Schema Conversion* module for creating the initial database in an engine 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
- A *Translation Module* takes this intermediate representation, and converts it to the target database language (the tool was originally designed with SQL in mind)
- The *Output Results & Information* module executes both the original Cypher on Neo4j, and the newly created query on the other chosen database(s). The execution times of the queries may be recorded, and the results of the queries may be written to a local file if requested. In the latter case, the tool can also verify that both the number of results and the results themselves match (this is useful for debugging).

The codebase is entirely Java. There are some additional features provided through the ANTLR tool, as described in *ANTLR Framework*. There are some script/.bat files that help the tool execute SQL on Postgres and pipe the results back into the tool.

Quick Usage

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

Argument	Description
<-s -schema>	Runs the tool for schema conversion.
<-t -translate>	Runs the tool for query translation.
<i>propertiesFile</i>	Path of the properties file (<i>c2s_props.properties</i>). See <i>Configuring the properties</i> for further information.
<i>targetDatabaseName</i>	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 <i>Translation of Queries</i> for further information.

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.

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 (for example, Postgres)
- Windows or Unix OS
- For large databases, a relatively powerful machine and available disk space (additionally, as much RAM as possible)

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
 - o 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 *Dealing with Cypher lists*)
- A folder for storing the results from both the Neo4j database, and the target database
 - o For example, create a folder called 'results', and then create two blank text files, *results\neo4jResults.txt* and *results\postgresResults.txt*
- A new Postgres database should also be created
 - o In Unix systems, this can usually be done by the command: *createdb <nameOfDB>*

Generating a dump file from a Neo4j database

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.

Windows

```
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"
```

Unix

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

The dump file should look something like:

```
begin
commit
begin
create (_0: `Local` { `mono_time`: "10", `name`: "omega", `node_id`: 1, `ref_count`: 1,
`sys_time`: -782642514, `type`: 4})
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)*.

Nodes

```
create (_1: `Global` { `name`: ["nginx"], `node_id`: 2, `sys_time`: -782642514, `type`: 2})
```

- `_1` becomes the id in the table of the relational schema
- 'Global' is the label of the node - this is stored in both the general *nodes* relation, but also, for speed reasons, in a separate relation named *global*
- The text within the curly brackets represents a JSON object of the properties

*NOTE: in this example, please note how the 'id' of the record is **different** from 'node_id', which is an internal property of the node from Neo4j itself. The record therefore will have two very similar but distinct columns.*

Edges

```
create (_12)-[: `LOC_OBJ` { `state`: 7}]->(_11)
```

- `_12` is the id identifier of the left node
- 'LOC_OBJ' is the type of relationship
- `_11` is the id identifier of the right node

- Edges may not have any properties, but they always have a type

It is a good idea to have this dump file in the folder created for this project, so that it isn't lost somewhere else in the filesystem.

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.

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

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.

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 black 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 *Graph to relational - theory*)
- The workspace folder will now contain four metafiles (see *Metafiles created during schema conversion* for more details)

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.

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*.

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

Function	Description
<i>array_unique(arr anyarray)</i>	Helper function for the <i>cypher_iterate</i> function (see row below). This function takes an array of any type as input, and returns an array of the same type, but with duplicate elements removed.
<i>cypher_iterate(integer[])</i>	Function for performing the iterate function designed for this project.
<i>doforeachfunc(integer[], field TEXT, newv TEXT)</i>	Function to help perform the semantics of Cypher's 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).

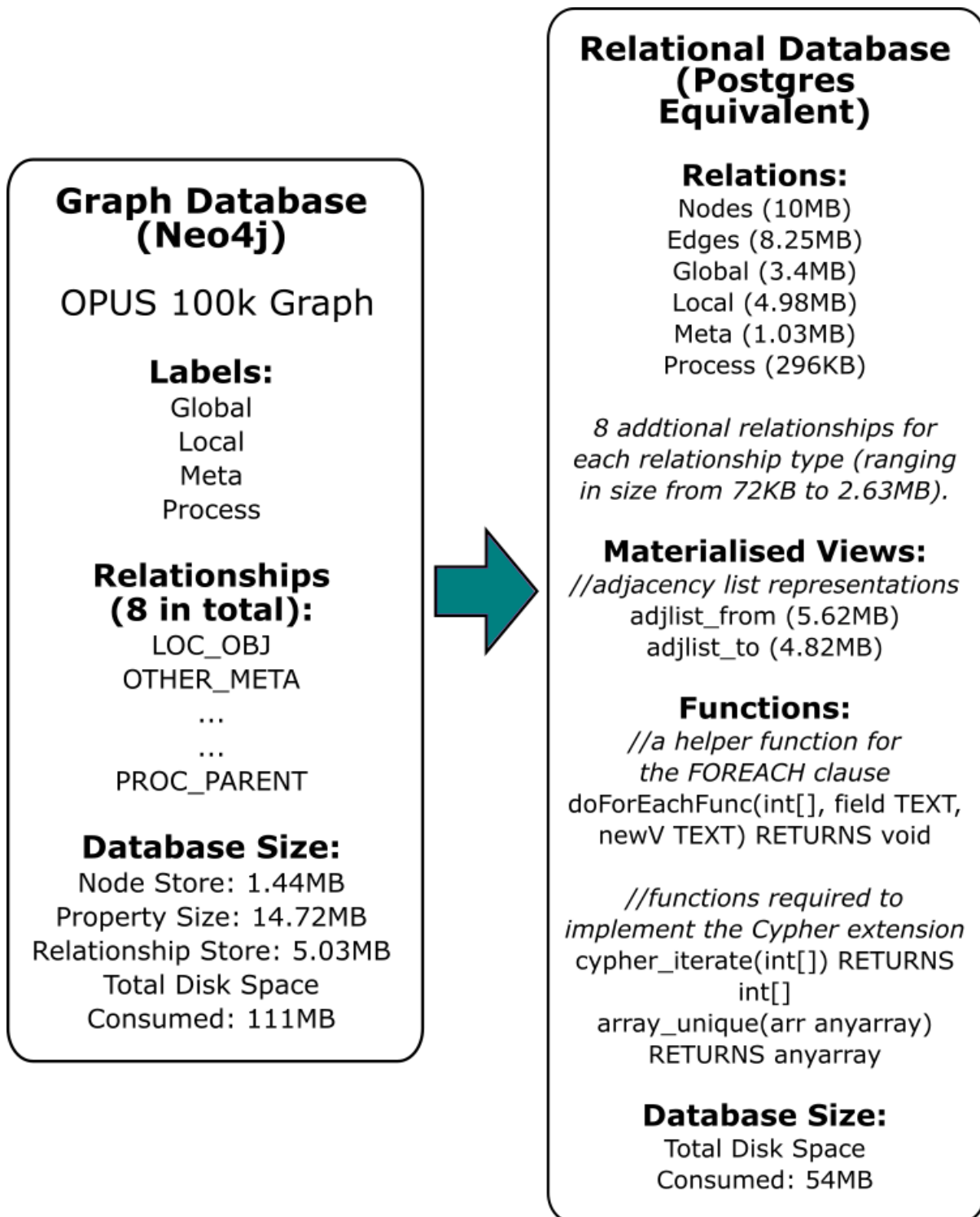


Figure 2 - Schema conversion outline.

Metafiles created during schema conversion

In total, four files are created at this stage.

File	Description
<i>meta_nodeProps.txt</i>	<p>Creation: all properties of the nodes are stored in this file.</p> <p>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.</p>
<i>meta_labelProps.txt</i>	<p>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.</p> <p>Example:</p> <pre>*country* name population dialling_code *city* name population state ave_house_price</pre> <p>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.</p> <p>See the method <i>getLabelMapping()</i> in <i>C2SMain.java</i> for further information. Information from this file is stored in the <i>labelProps</i> map (publicly accessible using <i>C2SMain.labelProps</i>)</p>
<i>meta_rels.txt</i>	<p>Creation: all the types of edges in the Neo4j graph are stored in this file.</p> <p>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).</p> <p>See the method <i>getLabelMapping()</i> in <i>C2SMain.java</i> for further information. Information from this file is stored in the <i>allRelTypes</i> map (publicly accessible using <i>C2SMain.labelProps</i>)</p>
<i>meta_labelNames.txt</i>	<p>Creation: all the names of the labels in the Neo4j graph are stored.</p> <p>Usage: another optimisation technique - helps the tool choose the best relation to use to complete the query correctly.</p>

Translation of Queries

With the graph schema now converted to a target database, queries may now be translated.

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 table below)
-

-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 may not have the right permissions to run with the tool. `chmod +x auto_run_pg.sh` should fix that problem.

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.id1 INNER
            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>

```

```

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> _

```

Query Translation

Tokenisation

The query is first tokenised using the ANTLR generated classes for the Cypher language. See *ANTLR Framework* for further details.

MATCH (n:Local)-->(m)-->(p) RETURN count(p) AS num_nodes;

becomes...

[match, (, n, :, local,), -, -, >, (, m,), -, -, >, (, p,), return, count, (, p,)]

NOTE: the alias num_nodes has been recorded by the tool but is omitted from the token list.

Intermediate Representation

The intermediate representation is in the form of a *DecodedQuery* object. This object is then referred to when translating to the target language (such as SQL). See *Workflow through the code* for more detailed descriptions.

Translation to SQL

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.

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 *Configuring the properties*).

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.

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 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:

Original Cypher input:

```
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

Original Cypher input:

```
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*' described the predicate itself:
 - o '*lt*' is the code for the < symbol, whilst the operand 0 is contained within the # symbols

Original Cypher input:

```
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)
 - o Extra care has to 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)

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

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 in the tool itself, 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.

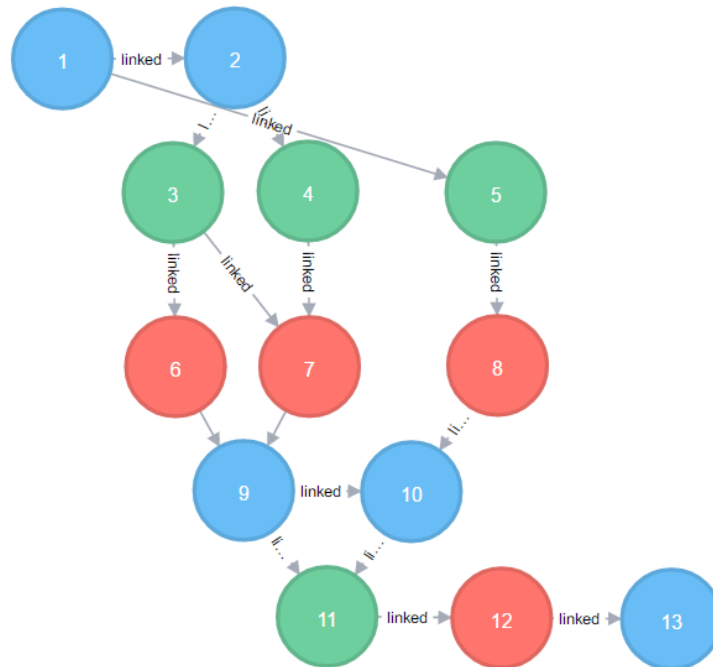


Figure 3 - Iterate Graph example.

In the graph, we map the colours in the following way: {blue=l1, green=l2, red=l3}. The figure shows a common pattern: BLUE->GREEN->RED->BLUE. Finding all the blue nodes at the end of this pattern 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;
```

This will execute on Postgres when translated to SQL, and will produce the following result:

9
9
9
10
13
13
13
13

In comparison, running just the **MATCH** part of the 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.

SQL equivalent of the Cypher above:

```

CREATE OR REPLACE FUNCTION loop_work(int[]) RETURNS int[] AS $$
  WITH a AS (
    SELECT n1.id AS a1, n2.id AS a2, e1.* FROM global n1
    INNER JOIN edges e1 on n1.id = e1.idl
    INNER JOIN local n2 on e1.idr = n2.id WHERE ( n1.id = ANY($1) )),
  b AS (
    SELECT n1.id AS b1, n2.id AS b2, e2.* FROM local n1
    INNER JOIN edges e2 on n1.id = e2.idl
    INNER JOIN meta n2 on e2.idr = n2.id),
  c AS (
    SELECT n1.id AS c1, n2.id AS c2, e3.* FROM meta n1
    INNER JOIN edges e3 on n1.id = e3.idl
    INNER JOIN global n2 on e3.idr = n2.id)
  SELECT array_agg(n01.id) FROM nodes n01, a, b, c
  WHERE a.a2 = b.b1 AND b.b2 = c.c1 AND n01.id = c.c2 AND a.a1 != b.b2
  AND a.a2 != c.c2 $$
LANGUAGE SQL;

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),
b AS (
  SELECT n1.id AS b1, n2.id AS b2, e2.* FROM local n1
  INNER JOIN edges e2 on n1.id = e2.idl
  INNER JOIN meta n2 on e2.idr = n2.id),
c AS (
  SELECT n1.id AS c1, n2.id AS c2, e3.* FROM meta n1
  INNER JOIN edges e3 on n1.id = e3.idl
  INNER JOIN global n2 on e3.idr = n2.id),
firstStep AS (
  SELECT (array_agg(n01.id)) AS list_ids FROM nodes n01, a, b, c
  WHERE a.a2 = b.b1 AND b.b2 = c.c1 AND n01.id = c.c2 AND a.a1 != b.b2
  AND a.a2 != c.c2),
collectStep AS (
  SELECT unnest((cypher_iterate(firstStep.list_ids))) AS zz from firstStep)

SELECT n01.node_id FROM nodes n01 INNER JOIN collectStep c ON n01.id = c.zz;

```

Workflow through the code

Below 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;
```

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

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)"
```

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 Iterate Semantics for more information.)

```

whereClause =
  "id(n) = 916 AND e.state in [5]"
returnClause = "c.name, e.state"
returnAlias = {HashMap@2938} size = 0
orderClause = "c.name desc"
latestOrderDirection = "desc"
skipAmount = -1
limitAmount = -1

```

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)"

```

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.

WhereClause object (*whereC*):

```
components = {ArrayList@3129} size = 2
0 = "id(n) = 916"
1 = "e.state in [5]"

whereMappings = {TreeMap@3133} size = 1
0 = {TreeMap$Entry@3393}
" id(n) = 916" -> "and"

clause = "id(n) = 916 AND e.state in [5]"
```

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*

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

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
```

Finally, the WHERE part of the Cypher query is decoded. The results are stored in a *WhereClause* object. This part of the method will **also** update the nodes stored in the *MatchClause* object with the correct properties.

NOTE: WhereClause has become deprecated - see Where Clauses for more information.

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).

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()*.

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.

```
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
```

ANTLR Framework

The current tool uses ANTLR v4² 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* project³.

The guide for setting up the parsers and lexers has been adapted from the ANTLR v4 ‘Getting Started’ guide⁴.

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.

- Create a new temporary folder, and add to it the latest Cypher.g4 grammar file
- Run the following command:
 - o `java -jar "C:\Program Files\Java\jdk1.8.0_101\lib\antlr-4.7-complete.jar" Cypher.g4`
 - o The grammar file was adjusted to work with the tool:
 - 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’
- Then run this command:
 - o `javac -classpath .;"C:\Program Files\Java\jdk1.8.0_101\lib\antlr-4.7-complete.jar" Cypher*.java`
- It is a good idea to test the installation at this point:
 - o `java -cp ".;C:\Program Files\Java\jdk1.8.0_101\lib\antlr-4.7-complete.jar" org.antlr.v4.gui.TestRig Cypher cypher -gui`
 - o Commands can now be typed into the console - ^Z will end the input and hopefully produce an image, such as the one in Figure 3

² <http://www.antlr.org/download.html>

³ <https://s3.amazonaws.com/artifacts.opencypher.org/M06/Cypher.g4>

⁴ <https://github.com/antlr/antlr4/blob/master/doc/getting-started.md>

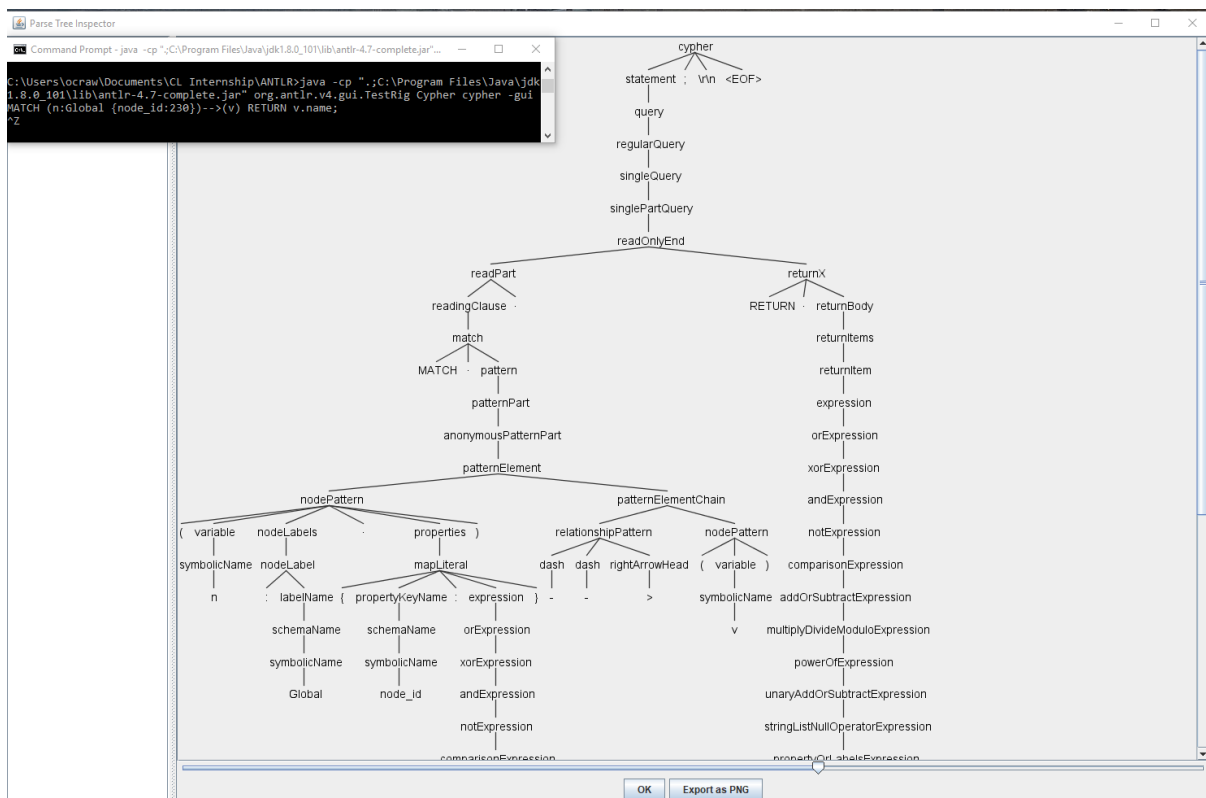


Figure 4 - Parse tree of a Cypher query with ANTLR.

Presuming the tool has successfully been installed, the following files should be copied into the source code:

- CypherBaseListener.java
- CypherLexer.java
- CypherListener.java
- CypherParser.java

Two classes rely on the ANTLR framework:

Class	Description
<i>CypherTokenizer.java</i>	Splits the original Cypher input into its tokens; parses the input through a parse tree to collect further information (see row below).
<i>CypherWalker.java</i>	This class extends <i>CypherBaseListener</i> and uses rule indexes as defined by the ANTLR framework to collect information such as: aliases, skip and limit amounts, and whether keywords like DISTINCT and OPTIONAL have been used.

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 adapted from full list of Cypher commands obtained from the Neo4j Cypher reference card: <https://neo4j.com/docs/cypher-refcard/current/>

NOTE: quantifiers common from regular expressions may be used below. As a quick reference, '' = zero or more times, '+' = one or more times, and '?' = zero or one time.*

NOTE: the collect semantics is currently functional and stable, but the output from the databases is different in terms of syntax used, and therefore the translation will flag up as incorrect. However, it should functionally be equivalent.

Type of Query

MATCH ... RETURN ...

Format of queries + examples

Format of queries:

```
MATCH (id)
RETURN id
```

```
MATCH (id : label)
RETURN DISTINCT id.property
```

```
MATCH (id : label {key:value})
RETURN id AS alias
```

```
MATCH (id {key:value})
RETURN id.propertyA, id.propertyB
```

```
MATCH (id : label)
RETURN count(id) AS alias
```

```
MATCH (id : label)
RETURN id.propertyA, CASE id.propertyB WHEN valueTest THEN
valueTrue ELSE valueFalse END
```

There should also be support in the tool for **multiple labels**, such as in the query:

```
MATCH (id:labelA:labelB)
RETURN count(id)
```

But this was not testable on the OPUS database I had available. The schema converter will convert multiple label nodes into one column on the relational database, containing a comma separated string of the labels.

Example queries for OPUS:

```
MATCH (a:Global)
RETURN count(a) AS all_global;
```



```
MATCH (a:Process {status:1})
RETURN a.sys_time, a.pid;
```

*ORDER BY, SKIP, and
LIMIT*

Format of queries:

```
MATCH (id {key:value})
RETURN id.propertyA
ORDER BY id.propertyB [ASC|DESC]?
[SKIP skipValue]? [LIMIT limitValue]?
```

Example queries for OPUS:

```
MATCH (a:Meta)
RETURN a.node_id
ORDER BY a.sys_time
SKIP 21 LIMIT 1;
```

*MATCH ... WHERE ...
RETURN*

Format of queries:

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

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

Example queries for OPUS:

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

Paths

Format of queries:

```
MATCH (idA)-->(idB)
WHERE idA.property = value
RETURN idB.property
```

```
MATCH (idA)-[:rel_type]->(idB)
RETURN idB.propertyA
ORDER BY idB.propertyB ASC
```

```
MATCH (idA)-[r : rel_type]-(idB)
RETURN idB.property AS node_prop_alias, r.property AS
rel_prop_alias
```

```
MATCH (idA {key:value})-[r : rel_type {key:value}]->()
RETURN count(r)
```

The pattern in the match clause shown above should extend for increasingly more connections:

```
MATCH (idA)-->(idB)-->(idC)-->(idD)
WHERE id(idD) < value
```

```
RETURN count(idA) AS alias;
```

Example queries for OPUS:

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

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

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

... WITH ... RETURN ...

Format for queries:

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

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

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

The translation of WITH queries requires some extra code that is inserted around a lot of the source code. It is therefore currently **not that stable** - WITH queries in Cypher should stick as closely as possible to one of the three patterns above, as shown in the example OPUS queries below.

The source code contains classes *With_Cypher.java* and *WithSQL.java* that can both be extended to fit more types of WITH queries.

Example queries for OPUS:

```
MATCH (a:Global)-[m]->(b:Local)--(c:Local)--(:Process)
WITH a, COUNT(m) AS glo_loc_count
WHERE glo_loc_count >= 4
RETURN a.node_id, glo_loc_count
ORDER BY glo_loc_count DESC;
```

```
MATCH (n)
WHERE 'Local' in labels(n) AND NOT exists(n.pid)
```

```
WITH n
MATCH (m:Global)-->(n)
WHERE id(m) > 900
RETURN n.node_id;
```

UNION and UNION ALL

Format of queries:

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

Variable length paths

Format of queries:

```
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)
```

A **direction** must be specified.

Example queries for OPUS:

```
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
```

Shortest Path queries

Format of queries:

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

Example queries for OPUS:

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

Functions: exists(), labels(), id()

Format of queries:

```
MATCH (idA)
WHERE exists(idA.property) AND exists(idA.otherProperty)
RETURN count(idA)
```

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

```
RETURN idA.propertyA
```

Example queries for OPUS:

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

Aggregate functions:
collect(), sum(),
avg(), min(), max().

Format of queries:

```
MATCH ... RETURN sum(id.property)
```

```
MATCH ... RETURN max(id.property)
```

Example queries for OPUS:

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

... WHERE property IN
predicate ...
any() function

Format of queries:

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

```
MATCH (idA)
WHERE any(someVar IN labels(idA) WHERE someVar IN [val1, val2, ..., valN])
RETURN count(idA)
```

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

Example queries for OPUS:

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

```
MATCH (a)
WHERE any(x in a.name WHERE x = 'Postgres')
RETURN count(a);
```

*ITERATE MATCH ...***Format of queries:**

```

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

```

Example queries for OPUS (note: only runs on SQL and not Neo4j):

```

ITERATE MATCH (a:Global)-->(b:Global)-->(c:Global)-->(d:Global)
LOOP d ON a COLLECT n
RETURN n.node_id;

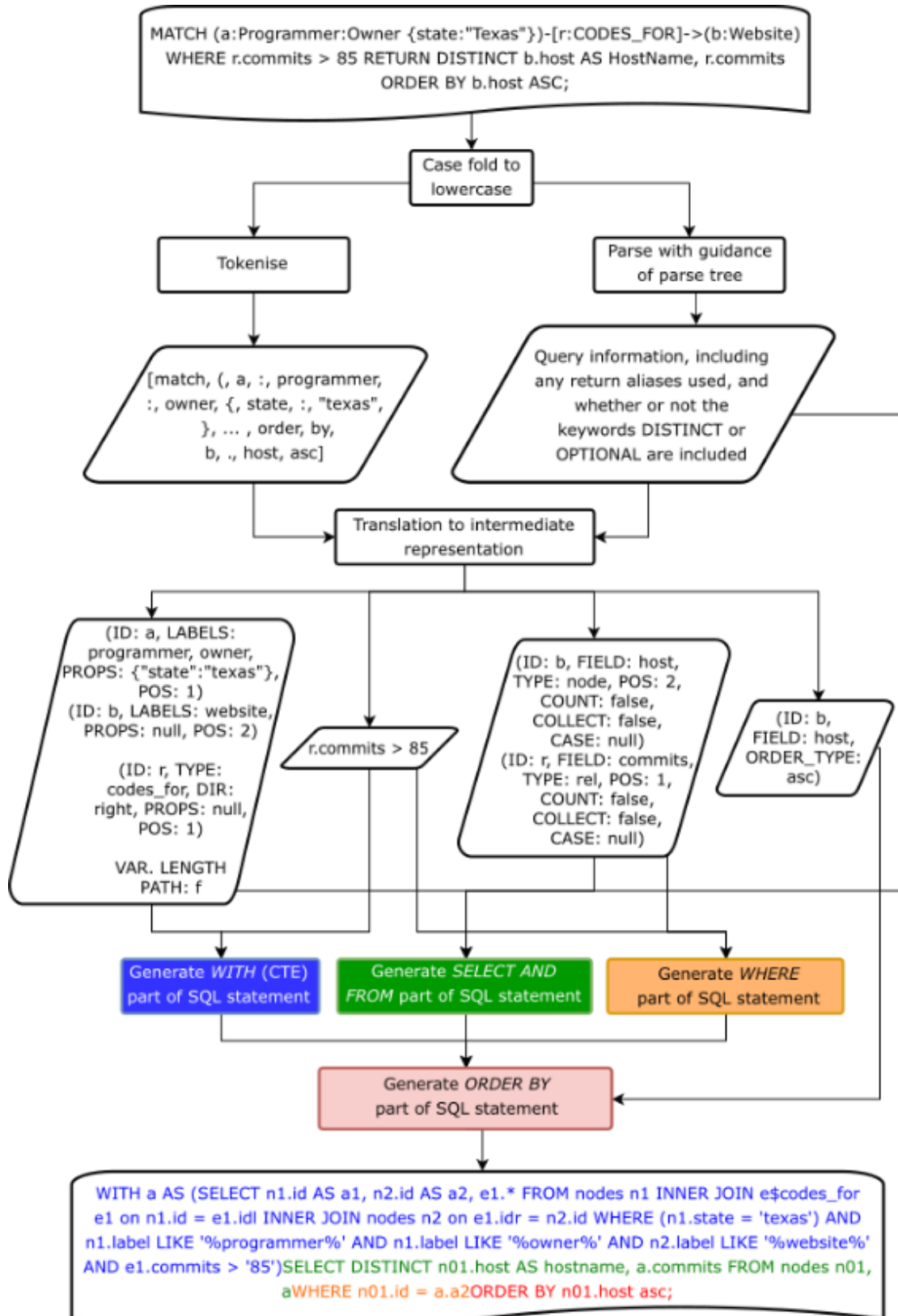
```

Known Issues

- The use of WITH is still quite constrained in its use.

Appendices

Appendix A - overview of translation of a Cypher query to SQL



Appendix B - current 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]-(i) 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)-->(n)
WHERE id(m) > 900 RETURN n.node_id;

```

```
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);
```