Cyp2SQL Documentation

Version 1.0

Last edited: 07/08/2017 14:00 by Oliver Crawford (ojc37@cam.ac.uk)

Table of Contents

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 another 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.

The codebase is entirely Java. There are some additional features provided through the ANTLR tool, as described in *ANTLR Framework*.

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.
propertiesFile	Path of the properties file (*c2s_props.properties*). See *Configuring the properties* for further information.
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 *Translation of Queries* 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.

NOTE: Javadoc documentation is available through the repo.

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

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
- The existing Neo4j graph to be placed into this folder (the graph should have a top-level folder – copy this over)
- 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
    - For example, create a folder called 'results', and then create two blank text files, *results\neo4jResults.txt* and *results\postgresResults.txt*
- Copy the file *c2s_props.properties* into this folder
    - This file can be found from the repo that should already have been downloaded

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-community-3.1.1/bin/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

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
neo4jSchema	Path of the dump file generated from the existing Neo4j graph (see *Generating a dump file from a Neo4j database*).
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 (see section blah…).
neo4jUser	Neo4j graph database username.
neoPW	Neo4j graph database password.
postgresUser	Postgres database username.
postgresPW	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 has now 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 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
array_unique(arr anyarray)	Helper function for the cypher_iterate 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.
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 Cypher's FOREACH keyword.

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

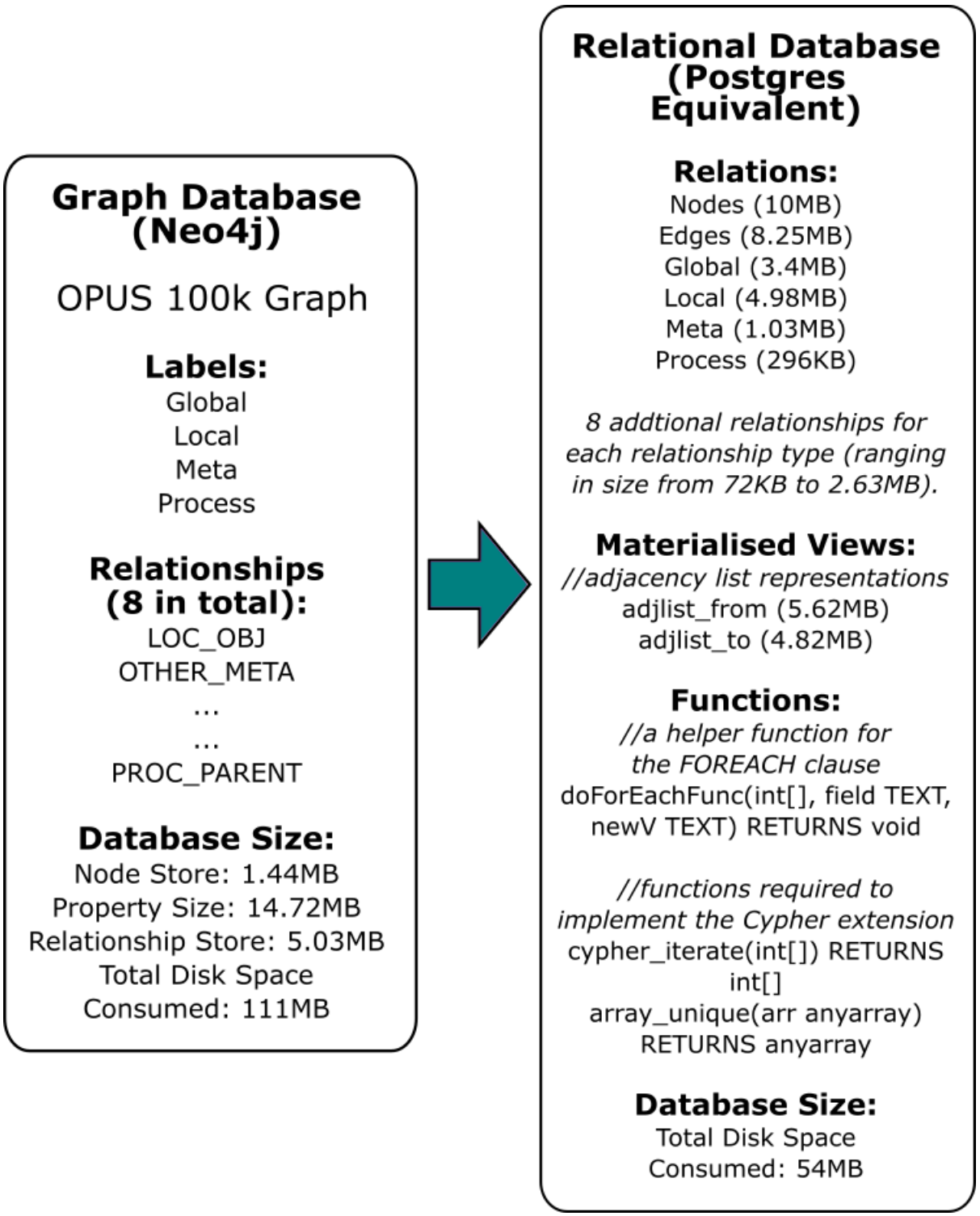

Figure 2 - Schema conversion outline.

Metafiles created during schema conversion

In total, four files are created at this stage.

File	Description
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.labelProps*
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.

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 C:\Users\ocraw\IdeaProjects\Cypher_SQL_Translation\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	**-r**
Debug interface.	***Run** as normal.*
This function allows the user at the command line to input queries 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.	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.

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).

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.

Whilst this leads to better object orientated code, the semantics of the query are broken in cases where the clause becomes long and mixes multiple operators (such as AND, OR, and NOT). The tool also does not currently handle brackets in where clauses.

This lack of total support for all types of where clauses may sound damaging, but it will mostly just require altering the SQL after the tool has completed to correctly add the brackets in the right places.

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.

- Create a new temporary folder, and add to it the latest Cypher.g4 grammar file
- Run the following command:
    - java -jar "C:\Program Files\Java\jdk1.8.0_101\lib\antlr-4.7-complete.jar" Cypher.g4
    - 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:
    - 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:
    - java -cp ".;C:\Program Files\Java\jdk1.8.0_101\lib\antlr-4.7-complete.jar" org.antlr.v4.gui.TestRig Cypher cypher -gui
    - 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

[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

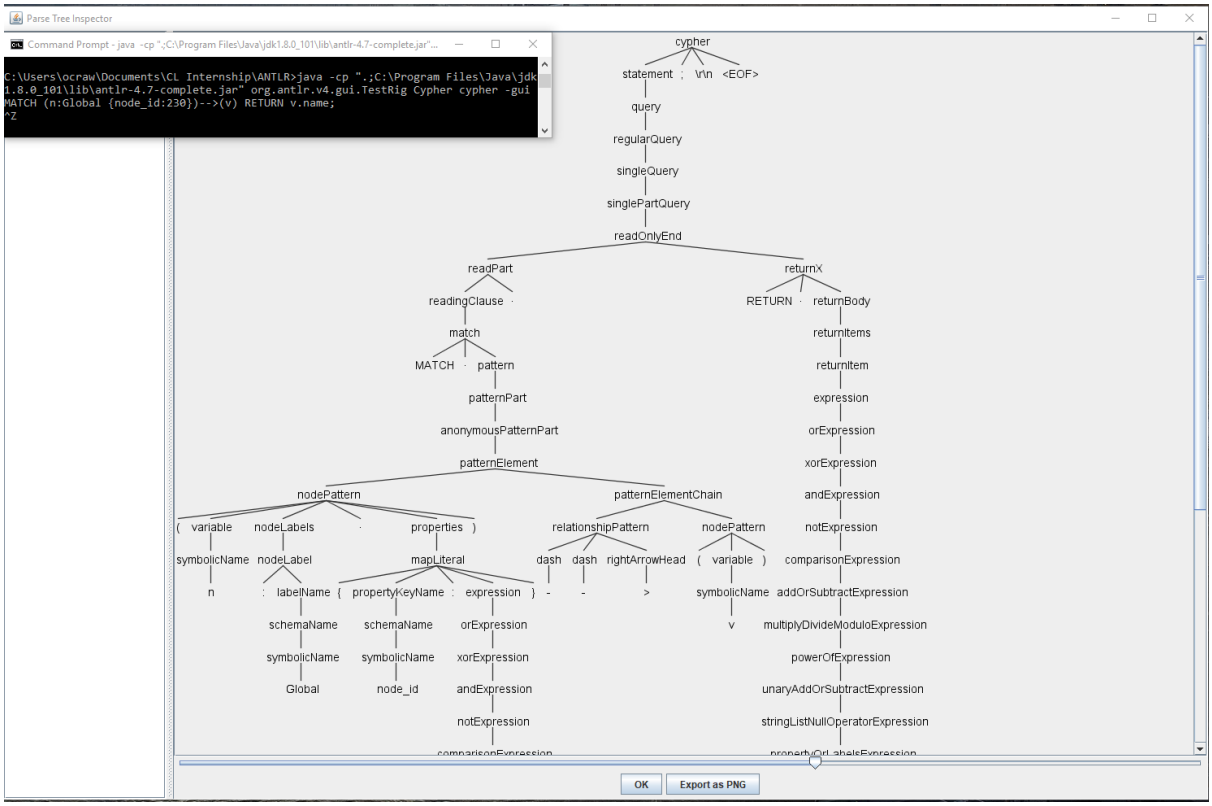

Figure 3 - Parse tree of a Cypher query with ANTLR.

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
CypherTokenizer.java	Splits the original Cypher input into its tokens; parses the input through a parse tree to collect further information (see row below).
CypherWalker.java	This class extends *CypherBaseListener* 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	Format of queries + examples
MATCH … RETURN …	**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*) **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);
… 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*) **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);

To be configured in later versions of the tool/documentation:

MATCH (n {node_id:620}) RETURN split(n.name[1], "db")[1]

select * from nodes where split_part(name[2], 'db', 2) in ('/entropy/saved-entropy.8')

Appendices

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