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:

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*.

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

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]-

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)"

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.

OrderClause object (orderC):

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

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.

Translation to SQL filter:

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).

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

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.