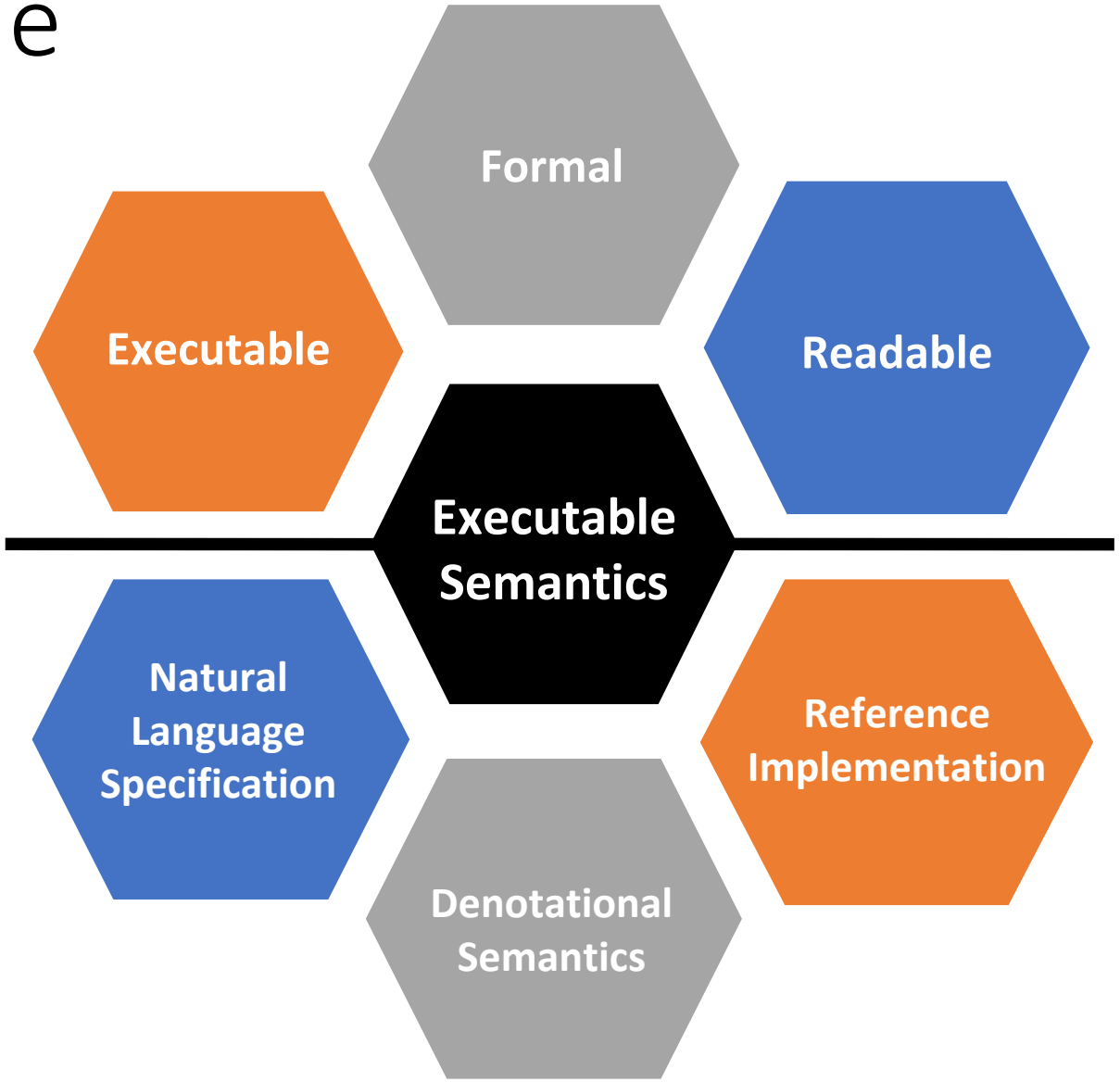
An Executable Semantics of Graph Query Language

12th LDBC TUC Meeting, Amsterdam

Jan Posiadała, Nodes and Edges

Three in one


Formal
Executable
Readable
Executable Semantics
Natural Language Specification
Reference Implementation
Denotational Semantics

Proof of concept: Cypher.PL

- an executable semantics
- of a declarative query language (Cypher)
- in the formal declarative language of logic (Prolog)
- as close to the semantics as possible
- as far from the implementation issues as possible
- a tool for collaborative language design
- and also an artifact of the design process

Why in Prolog?

- fully formalized declarative language
- built-in unification covers Cypher's pattern matching
- super-native representation of data with terms
- collects multiple matches (evident ambiguity)
- easy constraint verification
- native support for parsing (DCG)
- built-in meta-programming
- **ISO standard** (ISO/IEC 13211 by JTC 1/SC 22/WG 17)

Current source code status

Sociaal-**W**etenschappelijke **I**nformatica-Prolog (7.4.1)

Specifies semantics of Cypher 9+

- 1500 lines of code cover openCypher TCK test set
- reflects semantics ambiguity due to driving table order
- extra graphlet values support
  - binary and aggregation functions for union, intersection operations

```
create (a)-[:T]->(b)-[:T]->(c), (b)-[:T]->(d)
match p=()-->()
return agunion(p) as ug, agintersection(g) as ig
```

Cypher.PL basics

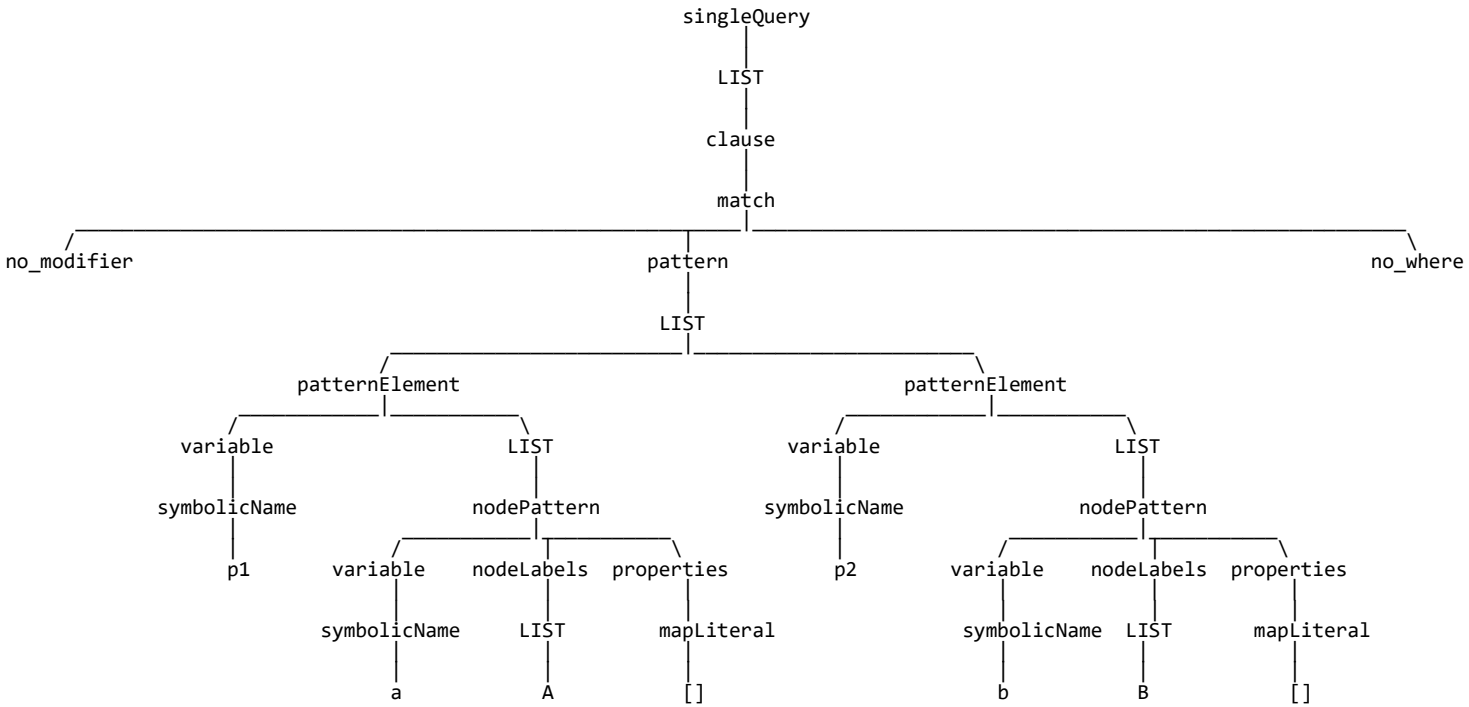
I. Abstract query

II. Evaluation environment: Table × Graph

III. Query evaluation

Abstract query representation

```
match p1=(a:A),p2=(b:B)
```



Syntax-sugar-free query

Base for semantics definition

Machine-oriented

- *Verbose*
- *Explicit*
- *Unambiguous*

Planner-friendly

- *Minimal ordering constraints*
- *Unique variable names*
- *Human-friendly*

Mainly for debugging, not a primary goal

Abstract query definition

```
single_query(singleQuery(Clauses)) :- maplist(clause,Clauses).

%%%%% Clauses %%%%%
clause(clause(Clause)) :- match(Clause).
%...
clause(clause(Clause)) :- with(Clause).
clause(clause(Clause)) :- return(Clause).

%%%%% Match %%%%%

match(Clause)
:-
Clause = match(MatchModifier,Pattern,Where),
match_modifier(MatchModifier),
pattern(Pattern),
where(Where).

match_modifier(no_modifier).
match_modifier(optional).
%match_modifier(mandatory). %in the future

pattern(pattern(PatternElements)) :- maplist(pattern_element,PatternElements).

pattern_element(patternElement(Variable,Patterns)) :-
variable(Variable),
patterns(Patterns).

patterns(Patterns)
:-
even_odd(Patterns, NodePatterns, RelationshipPatterns),
maplist(node_pattern, NodePatterns),
maplist(relationship_pattern, RelationshipPatterns).

node_pattern(nodePattern(Variable,NodeLabels,Properties)) :-
variable(Variable),
node_labels(NodeLabels),
properties(Properties).

node_labels(nodeLabels(Labels)) :- maplist(name,Labels).
```

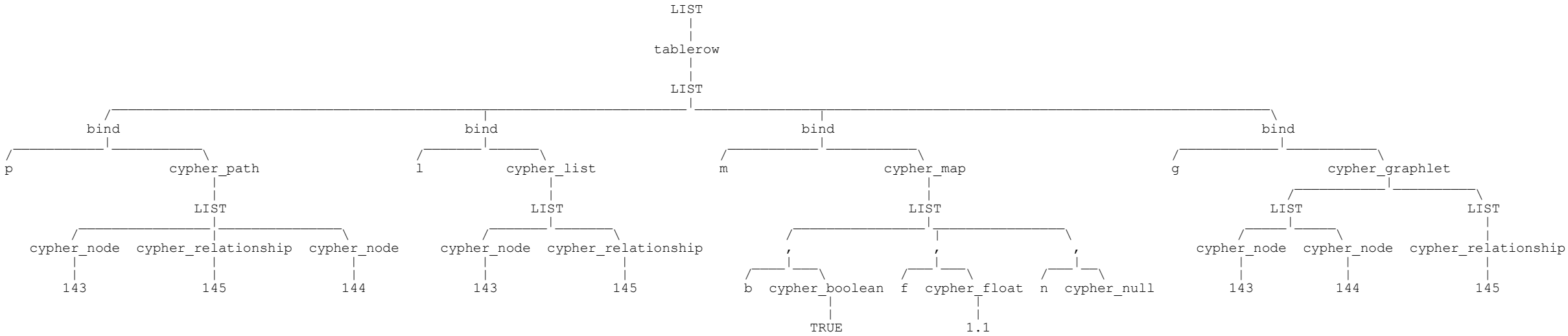
```
relationship_pattern(relationshipPattern(Variable
                                ,Direction
                                ,RelationshipTypes
                                ,RelationshipRange
                                        ,Properties))
:-
variable(Variable),
direction(Direction),
relationship_types(RelationshipTypes),
relationship_range(RelationshipRange),
properties(Properties).

direction(direction(left)).
direction(direction(right)).
direction(direction(both)).

relationship_types(relationshipTypes(Types)) :- maplist(name,Types).

relationship_range(relationshipRange(one_one)).
relationship_range(relationshipRange(L,U)) :-
(L=unlimited;integer(L)),%L >= 0,
(U=unlimited;integer(U)) %U >= 0
.
properties(properties(mapLiteral(Properties))) :- maplist(
[(propertyKeyName(PropertyName),Expression)]
>>
(schema_name(PropertyName),expression(Expression)),
        Properties).

where(where(Expression)) :- expression(Expression).
where(no_where).
```

Evaluation environment: Graph definition

```
%read (matching) “api”
node(NodeId,Graph) :- member(node(NodeId),Graph).
relationship(NodeStartId,RelationId,NodeEndId,Graph) :- member(relationship(NodeStartId,RelationId,NodeEndId),Graph).
property(NorRId,Key,Value,Graph) :- member(property(NorRId,Key,Value),Graph).
label(NodeId,Label,Graph) :- member(label(NodeId,Label),Graph).
type(RelationshipId,Type,Graph) :- member(type(RelationshipId,Type),Graph).

%write “api”
create_node(NodeId,Graph,ResultGraph) :-....
delete_node(NodeId,Graph,ResultGraph) :-....

set_label(NodeId,LabelName,Graph,ResultGraph) :-....
remove_label(NodeId,LabelName,Graph,ResultGraph) :-....

create_relationship(NodeStartId,NodeEndId,RelationshipType,RelationshipId,Graph,ResultGraph) :-....
delete_relationship(RelationshipId,Graph,ResultGraph) :-....

set_property(Id,Key,Value,Graph,ResultGraph) :-....
remove_property(Id,Key,Graph,ResultGraph) :-....
```

Evaluation environment: Table

```
create p=(a:A {int: 1, string: 'S', boolean : true, float: 1.1 })-[r:RELTYPE]->(b)
return p, [a,r] as l, {b : a.boolean, f : a.float, n : a.n} as m, gunion(p,p) as g
```



Evaluation environment: Table definition

```
is_row_of_table(TableRow) :-
is_list(TableRow),
maplist([bind(Name,Value)]>>(name(Name),cypher_value(Value)),TableRow).


%primitives:
%        cypher_null,
%        cypher_string(Value),
%        cypher_integer(Value),
%        cypher_float(Value),
%        cypher_boolean(Value),
%entities:
%        cypher_node(NodeId),
%        cypher_relationship(RelationshipId),
%        cypher_path(NodesRelationshipsAlernatingList),
%        extended with cypher_graphlet(CypherNodesList,CypherRelationshipsList)
%structures:
%        cypher_list(CypherValuesList),
%        cypher_map(NamesCypherValuesPairsList)
```

Query evaluation

```
eval_query(Graph, query(Clauses), env(ResultTable,ResultGraph))
:-
foldl(eval_clause, Clauses, env([[]],Graph), env(ResultTable,ResultGraph)).


eval_clause(clause(MatchClause), env(Table, Graph), env(ResultTable, Graph))
:-
MatchClause = match(MatchModifier, Pattern, Where),
findall(MatchTableRow,
       (
        member(TableRow, Table),
        eval_match(MatchModifier, Pattern, Where, Graph, TableRow, MatchTableRow)
       ),
       MatchResultTable).
```

Query evaluation: ambiguity handling

```
create ({num : 1}),({num : 2})
with *
match (n)
return (collect(n)[1]).num as nn
```

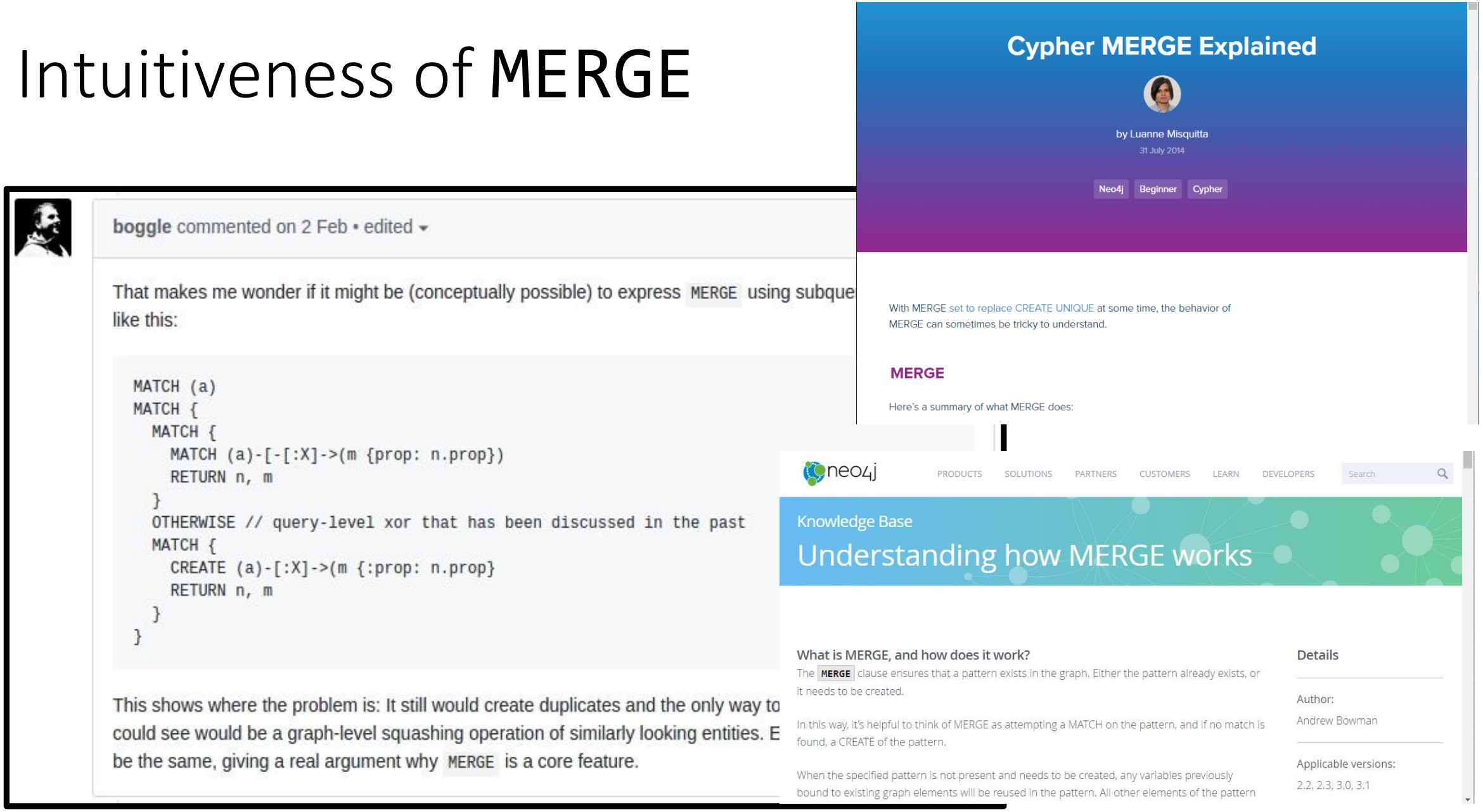
```
+--------+
|   nn   |
+--------+
|   1    |
+--------+
```

```
+--------+
|   nn   |
+--------+
|   2    |
+--------+
```

Query evaluation: ambiguity handling

```
eval_clause(clause(Clause),env(Table,Graph),env(ResultTable,ResultGraph))
 :-
 scall(set(Table), eval_clause_(Clause,Graph), environment_eq,
       env(ResultTable,ResultGraph)).

 scall(set(Set),Goal,EqGoal,Result)
 :-
 findall(PermutationResult,
        (permutation(Set, Permutation),
         call(Goal, Permutation, PermutationResult)),
         PermutationResults),
 gr_by(EqGoal, PermutationResults, ResultsGroups),
 member([Result|_], ResultsGroups).

 eval_clause_(Clause, Graph, Table, env(ResultTable, ResultGraph))
 :-
 %particular clause semantics definition

 environment_eq(env(Table1, Graph1), env(Table2, Graph2))
 :-
 permutation(Table1,Table2), isomorphic(Graph1,Graph2).
```

Intuitiveness of MERGE



boggle commented on 2 Feb • edited

That makes me wonder if it might be (conceptually possible) to express `MERGE` using subque
like this:

```
MATCH (a)
MATCH {
  MATCH {
    MATCH (a)-[-[:X]->(m {prop: n.prop})
    RETURN n, m
  }
  OTHERWISE // query-level xor that has been discussed in the past
  MATCH {
    CREATE (a)-[:X]->(m {:prop: n.prop}
    RETURN n, m
  }
}
```

This shows where the problem is: It still would create duplicates and the only way to
could see would be a graph-level squashing operation of similarly looking entities. E
be the same, giving a real argument why `MERGE` is a core feature.

Cypher MERGE Explained

by Luanne Misquitta

31 July 2014

Neo4j Beginner Cypher

With MERGE set to replace CREATE UNIQUE at some time, the behavior of MERGE can sometimes be tricky to understand.

MERGE

Here's a summary of what MERGE does:

neo4j PRODUCTS SOLUTIONS PARTNERS CUSTOMERS LEARN DEVELOPERS Search

Knowledge Base

Understanding how MERGE works

What is MERGE, and how does it work?

The `MERGE` clause ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.

In this way, it's helpful to think of MERGE as attempting a MATCH on the pattern, and if no match is found, a CREATE of the pattern.

When the specified pattern is not present and needs to be created, any variables previously bound to existing graph elements will be reused in the pattern. All other elements of the pattern

Details

Author:
Andrew Bowman

Applicable versions:
2.2, 2.3, 3.0, 3.1

Intuitiveness of MERGE

```
eval_merge(merge(patternElement(Variable,Patterns), MergeActions),
        Graph,
        TableRow,
        environment(ResultMatchTable,ResultMergeGraph))
:-
eval_clause(clause(match(no_modifier,pattern([patternElement(Variable,Patterns)]),no_where)),
         environment([TableRow],Graph),
         environment(ResultMatchTable,Graph)),
not(ResultMatchTable = []),
!,
convlist([onMatch(Set),Set]>>true, MergeActions,OnMatchActions),
foldl(eval_merge_actions,
      OnMatchActions,
      environment(ResultMatchTable,Graph),
      environment(ResultMatchTable,ResultMergeGraph)).
```

Intuitiveness of MERGE

```
eval_merge(merge(patternElement(Variable,Patterns), MergeActions),
        Graph,
        TableRow,
        environment(ResultCreateTable,ResultMergeGraph))
:-
eval_clause(clause(create(pattern([patternElement(Variable,Patterns)])))),
         environment([TableRow],Graph),
         environment(ResultCreateTable,ResultCreateGraph)),
convlist([onCreate(Set),Set]>>true, MergeActions,OnCreateActions),
foldl(eval_merge_actions,
      OnCreateActions,
      environment(ResultCreateTable,ResultCreateGraph),
      environment(ResultCreateTable,ResultMergeGraph)).
```

Broad standardization scope to support

- property graph query and update language
- … including errors definitions, raising and handling
- schema languages for property graphs
- languages interoperability
- session, transaction and concurrency model
- …

Proposal

A formal, readable, and executable semantics
can and should be
both a tool and an artifact
of language standardization.

Our latest activity

- Filip Murlak, Jan Posiadała, and Paweł Susicki. 2019. *On the semantics of Cypher's implicit group-by.* In Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages (DBPL 2019). ACM, New York, NY, USA, 59-69. DOI: https://doi.org/10.1145/3315507.333

- Upcoming: Filip Murlak, Jan Posiadała, and Paweł Susicki. 2019. *Executable semantics of graph query language. Cypher.PL case study.*

References

N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, M. Schuster, P. Selmer, and A. Taylor.

Formal Semantics of the Language Cypher.

CoRR, abs/1802.09984, 2018.

N. E. Fuchs.

Specifications are (preferably) executable.

Software Engineering Journal, 7(5):323–334, 1992.

R. A. Kowalski.

The relation between logic programming and logic specification.

Phil.Trans. of the Royal Society of London. 312(1522):345–361, 1984.