The Cypher Language 2017

Presentation to the LDBC Query Language Task Force
Neo Technology Cypher Language Group

Date of original presentation 3 July 2017
Submitted to DM32.2 13 July 2018

Neo4j Query Languages Standards and Research Team
The Cypher Language 2017

Pre-existing, agreed and planned features

Neo Technology Cypher Language Group  LDBC Query Language Task Force  3 July 2017
The problem space ...

How to concisely express in an “SQL-niche” declarative query language

Which graphs and (nested) tables are inputs to a query
The graphs and (nested) tables output by a query
The operations over graphs
The (optional, partial) schema of graphs
The storage of graphs (representations)
References to graphs (location, name) and their containers (stores)
... and its relationship to, or expression in, SQL

Does graph querying justify a special-purpose query language, distinct from SQL?

Or should SQL be extended to address the whole problem space?

If SQL is not extended to address the whole problem space then:

   How do we address the whole space?

Our view is that a “native” graph query language is necessary ...

and that it is also important to interoperate between the native language and SQL.

The most complete and widely used native graph query language is Cypher
The features of Cypher today

Property graph data model  Nodes and relationships have properties, and labels (types)
  Single, implicit global graph ("context graph"), in which paths can be processed

Ubiquitous visual pattern syntax  "Whiteboard friendly"
  Matching (identifying) subgraphs, creating/updating subgraphs, constraints/indices

Fully-featured query language  Reads, Updates, Schema definition
  Application-oriented type system — Scalars, Lists, Maps
  Result processing: Filtering, Ordering, Aggregation

By default, Cypher assumes heterogeneous data
Returns results as nested data (stored data limited to Scalar, List<Scalar>)
Use pipelining ("query parts") for query composition
Visual pattern syntax is not just for MATCHing

MATCH path=(a:Person)-[:KNOWS*2]->(b:Person) RETURN path

MATCH (a)-[r:LIKES]->(b) WHERE abs(a.age - b.age) < 5
WHERE NOT EXISTS (b)-[:MARRIED]-()

CREATE/MERGE (joe)-[:FRIEND]->(sue)

CREATE CONSTRAINT FOR (p:Person)-[:BORN_IN]->(c:City)

Visual pattern syntax is a pervasive feature of the graph query language

Enabling whiteboard-friendly, graph-oriented DML, DDL, and ultimately DCL, syntax
Composition with matching, aggregation, and sorting

// Top-down dataflow leads to natural query composition/chaining
// -- without introducing aliases and is similar to UNIX pipes

MATCH (p:Person)-[:KNOWS]->(friend:Person)
WITH p, count(friend) AS num_friends ORDER BY num_friends LIMIT 10
MATCH (p)-[:LIVES_IN]->(city:City)
RETURN p.name AS name, city.name AS city, num_friends

Top-down data flow enables visual query composition

Light-weight nested, correlated multi-row subqueries (like LATERAL in SQL)
The openCypher community: towards an open standard

In late 2015 Neo announced the openCypher initiative

- Apache-licensed grammar, ANTLR parser, TCK
- Open Cypher Improvements process based on Github issues/discussions
- Work has started on a formal specification of Cypher (denotational semantics) by University of Edinburgh
Governed by the openCypher Implementers Group

In 2017 two face-to-face openCypher Implementers Meetings have taken place.

Regular openCypher Implementers Group virtual meetings scheduled through to October.

Consensus-based governance: open to all, but implementers “at the heart of the consensus”
Cypher implementations

Cypher is used as the graph query language of four commercial/OSS databases

- Neo4j Enterprise Server, SAP HANA Graph, AgensGraph/Postgres, and RedisGraph

There are other databases/query engines in gestation or in the research community

- Memgraph, Ingraph, Scott Tiger, Cypher for Apache Spark, Graphflow ...

There are several other projects or tools that use Cypher

- IDEA plugin from Neueda, language parsers, editors, GraphQL Cypher directives, ...
Cypher 2017 + PGQL → “CyQL”

SQL

SQL/PGQ

SQL Graph Representations + Graph Functions

Cypher 2016

PGQL 1.0

LDBC QL TF

desired features
Features in active design or in adoption

**Query Composition and Set Operations**

**Nested Subqueries** *(Scalar, Existential, Correlated, Updating, …)*

**Map Comprehensions for working with nested data**

**Additional Constraints**

**Configurable *morphism**

**Path expressions and path patterns**

**Support for Multiple Graphs (graph-returning query composition)**
Path Expressions and Path Patterns

PATH PATTERN unreciprocated_love = (a)-[:LOVES]->(b)
    WHERE NOT EXISTS { (b)-[:LOVES]->(a) }
MATCH (you)-/-unreciprocated_love*/->(someone)

Compared to GXPath

Compared to Regular Expressions With Memory (REMs)
Use-cases for multiple-graph support

Data integration  Combining multiple data sources
Security  Graph views
Visualization  Returning graphs to the client
Time-based comparison  Snapshots/Versioned Graphs, Deltas
Fraud-ring detection  Graphlet results (multiple matching subgraphs)
Composition  Function chains of queries and other graph functions
Summarization  Show an abstracted (aggregated and/or simplified) graph
Cypher support for working with multiple graphs

Introduce globally addressable graphs with a graph URI scheme: `graph://...`

Enable naming and referring to graphs produced by earlier stages of a query/session

Introduce query context read and write graphs for ease of use and composition.

Create and amend graphs by emitting commutative updates into a target graph.

Define graph query composition via pipelining inside the language or outside the language (e.g. composing queries with other functions over graphs using an API).
Cypher Today: Queries (not yet) closed over graphs

Read queries take a graph $G$, yielding a nested tabular result (relational sub-graph view)

Lists and maps for parameters and results

Cypher transforms graphs to nested tables

$G + \text{(Nested)} \ T \rightarrow \text{(Nested)} \ T'$

Write queries take a graph $G$, and modify it (implicitly resulting in $G'$), but can only return a nested tabular result
Cypher Today: Queries (not yet) closed over graphs

Queries that only return tabular data do not allow graph-level query composition:

- Impossible to identify or retrieve (use as an input) $G'$ distinct from $G$
- Therefore neither read nor write queries can be composed.

Transform Table

19
Cypher this summer: Queries closed over graphs

Natively handling multiple graphs requires adding **graph references**

Needed to refer to existing graphs or those produced as intermediary results

Enables seamless query composition / functional chaining.
Compositional queries

Accept the same type as they return

That type at its most general could be

- a tuple of graph references plus
- a tuple of table references

Cypher 2017 is focussed on a tuple of graphs and one (nested) table

This adds graphs in, graphs out to Cypher 2017

Cypher pipelining (WITH) is an existing compositional mechanism that is extended
Multiple Graphs Syntax and Pipeline Composition

WITH a, b GRAPHS g1, g2 // Normal Cypher composition and selection

FROM GRAPH <name> // Sets source and target graph for the following statements
    AT 'graph://...' // Resolves physical address

INTO NEW GRAPH <name> // Sets target graph for the following statements
    AT 'graph://...' // Resolves physical address

RETURN a, b GRAPHS g1, g2 // Returns table and graphs

WITH a, b GRAPHS g1, g2 ... // first query

WITH GRAPHS g3, g4 ... // second query over first query

RETURN c, d GRAPHS g5 // third query over second query over first query
Example 1

FROM GRAPH foo AT 'graph://my-graph'

MATCH (a:Person)-[r:KNOWS]->(b:Person)
MATCH (a)-[:LIVES_IN]->(c:City)<-[:LIVES_IN]-(b)
INTO NEW GRAPH berlin
CREATE (a)-[:FRIEND]->(b) WHERE c.name = "Berlin"
INTO NEW GRAPH santiago
CREATE (a)-[:FRIEND]->(b) WHERE c.name = "Santiago"
RETURN c.name AS city, count(r) AS num_friends GRAPHS berlin, santiago
LDBC Example 1

```
FROM GRAPH AT "graph://social-network" // Set scope to whole social network
MATCH (a:Person)-[:KNOWS]->(b:Person)-[:KNOWS]->(c:Person)
   WHERE NOT (a)--(c)
INTO NEW GRAPH recommendations // Create a temporary named graph
CREATE (a)-[:POSSIBLE_FRIEND]->(c) // Containing existing nodes and new rels
FROM GRAPH recommendations // Switch context to named graph
MATCH (a:Person)-[e:POSSIBLE_FRIEND]->(b:Person)
RETURN a.name, b.name, count(e) AS cnt // Tabular output, and ...
   ORDER BY cnt DESC
   GRAPHS recommendation // ... graph output!
```
FROM GRAPH AT "graph://social-network" // Set scope to whole social network

MATCH (a:Person)-[:IS_LOCATED_IN]->(c:City),
    (c)->[:IS_LOCATED_IN]->(co:Country),
    (a)-[e:KNOWS]-(b)

INTO NEW GRAPH sn_updated

CREATE (a)-[e]-(b) // Add previous matches to new graph
    SET a.country = cn.name // Update existing nodes

FROM GRAPH sn_updated

MATCH (a:Person)-[e:KNOWS]->(b:Person)

WITH a.country AS a_country, b.country AS b_country, count(a) AS a_cnt, count(b) AS b_cnt,
    count(e) AS e_cnt

INTO NEW GRAPH rollup

MERGE (:Persons {country: a_country, cnt: a_cnt})-[:KNOW {cnt: e_cnt}]->(:Persons {country: b_country, cnt: b_cnt})

RETURN GRAPH rollup // Return graph output
LDBC Example 3

FROM GRAPH AT 'graph://social-network'

MATCH (a:Person)-[e]->(b:Person),
   (a)-[:LIVES_IN]->()-[:IS_LOCATED_IN]-(c:Country {name: 'Sweden'}),
   (b)-[:LIVES_IN]->()-[:IS_LOCATED_IN]-(c)

INTO GRAPH sweden_people AT './swe'

CREATE (a)-[e]->(b)

MATCH (a:Person)-[e]->(b:Person),
   (a)-[:LIVES_IN]->()-[:IS_LOCATED_IN]-(c:Country {name: 'Germany'}),
   (b)-[:LIVES_IN]->()-[:IS_LOCATED_IN]-(c)

INTO GRAPH german_people AT './ger'

CREATE (a)-[e]->(b)

RETURN *

FROM GRAPH sweden_people

MATCH p=(a)--(b)--(c)--(a) WHERE NOT (a)--(c)

INTO GRAPH swedish_triangles

RETURN count(p)

GRAPH swedish_triangles

// Set scope to whole social network

// Create a persistent graph “sn/swe”

// Containing persons and knows rels

// Create a persistent graph “sn/ger”

// Containing :Person nodes and :KNOWS rels

// Graph output (no tabular)

// Start query on Sweden people graph

// Create a temporary graph

// Tabular and graph output
Many possible syntactical transformations

FROM GRAPH foo
MATCH ()-->(())-->(n)
MATCH (n)-->(())-->()
INTO GRAPH bar
CREATE *

<==>
FROM GRAPH foo
MATCH (a)-[r1]->(b)-[r2]->(c)
MATCH (c)-[r3]->(d)-[r4]->(e)
INTO GRAPH bar
CREATE (a)-[r1]->(b)-[r2]->(c)
CREATE (c)-[r3]->(d)-[r4]->(e)
Not 2 to 3, but 2 into 1

openCypher
PGQL
QL TF GQL
...

Cypher 2017 + PGQL → “CyQL”

SQL Graph Representations + Graph Functions

SQL/PGQ

LDBC QL TF
desired features
Appendix  History and survey of Cypher
The inception of property graphs

Real-world use cases
- Content management, recommendations, IAM, life sciences
- Graph first, transactional online processing (not just static analysis)
- Heterogenous data
- Quickly evolving (sometimes no) schema
- High traversal performance
- Results consumable by existing applications and libraries

Approaches
- Relational systems did not fit well at the time: Expressivity and performance
- Similarly with RDF at that time: Complexity and performance
- Initial prototype: Fast graph traversal engine
Initial Motivation for Cypher “The SQL of property graphs”

Declarative query language Use patterns to retrieve nodes, relationships, paths
Read and write/update queries

Heterogenous data language using optional schema
Static and dynamic typing should be possible

Visual language
Graph patterns (ASCII-art) (startNode)-[relationship:TYPE]->(endNode)

Applications language
From nothing to something: "top-down dataflow" (or "reverse SQL") order
Returns (nested) tabular data for integrating with existing applications
Covers majority of use-cases for manually implemented traversals
The features of Cypher today

Property graph data model  Nodes and relationships have properties, and labels (types)
  Single, implicit global graph ("context graph"), in which paths can be processed

Ubiquitous visual pattern syntax  "Whiteboard friendly"
  Matching (identifying) subgraphs, creating/updating subgraphs, constraints/indices

Fully-featured query language  Reads, Updates, Schema definition
  Application-oriented type system — Scalars, Lists, Maps
  Result processing: Filtering, Ordering, Aggregation

By default, Cypher assumes heterogeneous data
Returns results as nested data (stored data limited to Scalar, List<Scalar>)
Use pipelining ("query parts") for query composition
Property Graph Data Model  openCypher, Neo4j > 2.0

Nodes with multiple labels ("roles") and multiple properties
Relationships with single type (label) and multiple properties
Grounded in >10 years experience with real world use cases from large enterprises
Next big step: Support for multiple graphs

Meta-properties (mainly for per-property auth)
Multi-properties (mainly covered by list properties)
**Visual Pattern Matching: Reads**

MATCH (a)-[r:FRIEND]->(b)
RETURN * // RETURN a, r, b

// Paths variables

MATCH p=(a:Person)-[:FRIEND*..3]->(b:Person),
    (a)-[:LIVES_IN]->(c:City)<-[[:LIVES_IN]]-(b)
WHERE abs(a.born - b.born) < 5
RETURN p
Visual Pattern Matching: Updates

CREATE (joe:Person {name: 'Joe'})
CREATE (joe)-[:FRIEND]->(sue)

MERGE (jack:Employee {name: 'Joe'})
  ON CREATE SET jack.created = timestamp()
  ON MATCH SET jack.updated = timestamp()
SET jack.salary = $newSalary

SET node:Label
REMOVE node:Label
SET node.prop = 42
REMOVE node.prop

DELETE node, rel
// Post-processing
MATCH (a)-[r:LIKES]->(b) WHERE abs(a.age - b.age) < 5
WHERE NOT EXISTS (b)-[:MARRIED]-()
RETURN b ORDER BY r.how_much

// Schema constraints and Index creation
// CIP2016-12-14-Constraint-syntax: an earlier syntax in Neo4j
ADD CONSTRAINT person_detail
FOR (p:Person)-[:BORN_IN]->(c:City)
REQUIRE exists(p.name)
AND p.born >= c.founded
Visual pattern syntax is not just for MATCHing

MATCH (a)-[r:LIKES]->(b) WHERE abs(a.age - b.age) < 5
NOT EXISTS (b)-[:MARRIED]-()
PATH PATTERN co_author =
    (a)-[:AUTHORED]->(:Book)<-[:AUTHORED]-(b)
CREATE/MERGE (joe)-[:FRIEND]->(sue)
FOR (p:Person)-[:BORN_IN]->(c:City)

Visual pattern syntax is a pervasive feature of the graph query language
Enabling whiteboard-friendly, graph-oriented DML, DDL, and ultimately DCL, syntax
Query syntax order and composition

// From nothing to something: top-down dataflow
MATCH (n) RETURN *

// Leads to natural query composition/chaining
// -- without introducing aliases and similar to UNIX pipes
MATCH (p:Person)-[:KNOWS]->(friend:Person)
WITH p, count(friend) AS num_friends ORDER BY num_friends LIMIT 10
MATCH (p)-[:LIVES_IN]->(city:City)
RETURN p.name AS name, city.name AS city, num_friends

Top-down data flow enables visual query composition

Light-weight nested, correlated multi-row subqueries (like LATERAL in SQL)
“SQL-alien” features

// Querying heterogeneous data
// - This arises during early exploration/integration of
disparate data sets
// - Facilitates fast-paced microservice architectures

MATCH (a:Document)-[:FROM]-(s:Source)
RETURN a.id, a.score, properties(s)

MATCH (anything)-[:FROM]-(:Source)
RETURN properties(anything)

// Returning nested data is common requirement for web apps

MATCH (person:Person {userId: $user})-[::ADDRESS]->(address)
RETURN person {firstName, lastName, id: $user,
              address {streetAddress, city, postalCode} }