openCypher

New Directions in Property Graph Querying

EDBT/ICDT 2018 Tutorial

Petra Selmer
openCypher Language Group @ Neo4j
PhD @ Birkbeck, University of London

Martin Junghanns
Cypher for Apache Spark @ Neo4j
PhD Student @ University of Leipzig
Intended Audience

Cypher and proposed extensions (Cypher 10 onwards)

People that want to learn about
The Property Graph Data Model
The Cypher Graph Query Language
Future extensions of the Cypher language

Cypher for Apache Spark

People that want to
Learn about Apache Spark
Know how to translate Cypher queries into Spark SQL programs
See Cypher’s multiple graph extensions and query composition in action
Implement Cypher on another backend besides Apache Spark
Agenda

The property graph data model
The Cypher query language
Evolving Cypher through the openCypher project
Cypher extensions (Cypher 10 and onwards)
  - Multiple graphs and query composition
  - Complex path patterns
  - Configurable pattern-matching semantics

Cypher for Apache Spark
  - Technical Architecture
  - Cypher to Spark SQL
  - Programming API
  - Demonstration
About me

Member of the Cypher Language Group

Design new features for Cypher
Coordinate the openCypher project

Engineer at Neo4j

Work on the Cypher Features Team

PhD in flexible querying of graph-structured data, Birkbeck, University of London

Approximation and relaxation of regular path queries
Work and content thanks to...

Colleagues and fellow authors in the openCypher Group at Neo4j

Stefan Plantikow  Tobias Lindaaker  Alastair Green  Mats Rydberg  Andrés Taylor

Thanks also to Neo4j for the use of some of the slides
Formal semantics

Collaboration with the University of Edinburgh, UK

Nadime Francis*, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Martin Schuster

*Now at Université Paris-Est, France

Defining the formal semantics of Cypher

Core read-only fragment published

Upcoming paper at SIGMOD 2018:
Cypher: An Evolving Query Language for Property Graphs
The property graph data model
What is a property graph?
Property graph

Underlying construct is a graph

Four building blocks:

- Nodes (synonymous with vertices)
- Relationships (synonymous with edges)
- Properties (map containing key-value pairs)
- Labels

https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc
Property graph

Node

- Represents an entity within the graph
- Has zero or more *labels*
- Has zero or more *properties*
  (which may differ across nodes with the same label(s))
Property graph

Node
- Represents an entity within the graph
- Has zero or more labels
- Has zero or more properties
  (which may differ across nodes with the same label(s))

Relationship
- Adds structure to the graph
  (provides semantic context for nodes)
- Has one type
- Has zero or more properties
  (which may differ across relationships with the same type)
- Relates nodes by type and direction
- Must have a start and an end node
Property graph

Node

- Represents an entity within the graph
- Has zero or more labels
- Has zero or more properties (which may differ across nodes with the same label(s))

Relationship

- Adds structure to the graph (provides semantic context for nodes)
- Has one type
- Has zero or more properties
- Relates nodes by type and direction
- Must have a start and an end node

Property

- Name-value pair (map) that can go on nodes and relationships
- Represents the data: e.g. name, age, weight etc
- String key; typed value (string, number, bool, list)
Formal definition

3.1 Values

We consider three disjoint sets $\mathcal{K}$ of property keys, $\mathcal{N}$ of node identifiers and $\mathcal{R}$ of relationship identifiers (ids for short). These sets are all assumed to be countably infinite (so we never run out of keys and ids). For this presentation of the model, we assume two base types: the integers $\mathbb{Z}$, and the type of finite strings over a finite alphabet $\Sigma$ (this does not really affect the semantics of queries; these two types are chosen purely for illustration purposes).

The set $\mathcal{V}$ of values is inductively defined as follows:

- Identifiers (i.e., elements of $\mathcal{N}$ and $\mathcal{R}$) are values;
- Base types (elements of $\mathbb{Z}$ and $\Sigma^*$) are values;
- $true$, $false$ and $null$ are values;
- list() is a value (empty list), and if $v_1, \ldots, v_m$ are values, for $m > 0$, then list$(v_1, \ldots, v_m)$ is a value.
- map() is a value (empty map), and if $k_1, \ldots, k_m$ are distinct property keys and $v_1, \ldots, v_m$ are values, for $m > 0$, then map$((k_1, v_1), \ldots, (k_m, v_m))$ is a value.
- If $n$ is a node identifier, then path$(n)$ is a value. If $n_1, \ldots, n_m$ are node ids and $r_1, \ldots, r_{m-1}$ are relationship ids, for $m > 1$, then path$(n_1, r_1, n_2, \ldots, n_{m-1}, r_{m-1}, n_m)$ is a value. We shall use shorthands $n$ and $n_1 r_1 n_2 \ldots n_{m-1} r_{m-1} n_m$.

3.2 Property graphs

Let $\mathcal{L}$ and $\mathcal{T}$ be countable sets of node labels and relationship types, respectively. A property graph is a tuple $G = (N, R, src, tgt, \iota, \lambda, \tau)$ where:

- $N$ is a finite subset of $\mathcal{N}$, whose elements are referred to as the nodes of $G$.
- $R$ is a finite subset of $\mathcal{R}$, whose elements are referred to as the relationships of $G$.
- src: $R \rightarrow N$ is a function that maps each relationship to its source node.
- tgt: $R \rightarrow N$ is a function that maps each relationship to its target node.
- $\iota$: $(N \cup R) \times \mathcal{K} \rightarrow \mathcal{V}$ is a function that maps a (node or relationship) identifier and a property key to a value.

It is assumed that $\iota$ is a total function but that its “non-null support” is finite: there are only finitely many $j \in (N \cup R)$ and $k \in \mathcal{K}$ such that $\iota(j, k) \neq null$.

- $\lambda$: $N \rightarrow 2^\mathcal{L}$ is a function that maps each node id to a finite (possibly empty) set of labels.
- $\tau$: $R \rightarrow \mathcal{T}$ is a function that maps each relationship identifier to a relationship type.

Refer to the papers for more details.
Compared against other graph data models
Other graph-structured models

Researched for decades

  Plenty of variations of simple directed labelled graphs
  Hypergraph model

  *Survey of Graph Database Models*, R. Angles, and C. Gutierrez

Semi-structured data models:

  XML

  OEM: *Object exchange across heterogeneous information sources*,
  Y. Papakonstantinou, H. Garcia-Molina, and J. Widom
RDF Data Model

Triple:

*Subject*: describes a resource; modelled as a node

*Predicate*: this is a property of the resource; modelled as an edge

*Object*: the value of the property; modelled as a node
When and why is it useful?
Relational vs. graph models
Relationship-centric querying

Query complexity grows with need for JOINs

Graph patterns not *easily* expressible in SQL

  Recursive queries

  Variable-length relationship chains

Paths cannot be returned natively
The topology is as important as the data...
Data integration

CONSUMER DATA

PRODUCT DATA

PAYMENT DATA

SOCIAL DATA

SUPPLIER DATA
Real-world usage
Use cases

Impact Analysis

Logistics and Routing

Recommendations

Access Control

Fraud Analysis

Social Network
Examples of graphs in industry

Organization

Identity & Access

Network & IT Ops

Figure 5-7. Talent.net graph enriched with WORKED_WITH relationships

Figure 5-8. Access control graph

Figure 3-5. Example graph for the data center deployment scenario
Data centre dependency network

- Nodes model applications, servers, racks, etc
- Relationships model how these entities are connected
- Impact analysis

Figure 3-5. Example graph for the data center deployment scenario
Some well-known use cases

NASA

Knowledge repository for previous missions - root cause analysis

Panama Papers

How was money flowing through companies and individuals?
The Cypher query language
Introducing Cypher

Declarative **graph pattern matching** language

SQL-like syntax

- DQL for reading data
- DML for creating, updating and deleting data
- DDL for creating constraints and indexes
Graph patterns

(:Person { name: "Dan"}) -[:LOVES]-> (:Person { name: "Ann"})
Searching for (matching) graph patterns

MATCH (:Person { name:"Dan"}) -[:LOVES]-> ( whom ) RETURN whom
DML: Creating and updating data

// Data creation and manipulation
CREATE (you:Person)
SET you.name = 'Jill Brown'
CREATE (you)-[:FRIEND]->(me)

// Either match existing entities or create new entities.
// Bind in either case
MERGE (p:Person {name: 'Bob Smith'})
  ON CREATE SET p.created = timestamp(), p.updated = 0
  ON MATCH SET p.updated = p.updated + 1
RETURN p.created, p.updated
DQL: reading data

// Pattern description (ASCII art)
MATCH (me:Person)-[:FRIEND]->(friend)
// Filtering with predicates
WHERE me.name = 'Frank Black'
AND friend.age > me.age
// Projection of expressions
RETURN toUpper(friend.name) AS name, friend.title AS title
// Order results
ORDER BY name, title DESC

Multiple pattern parts can be defined in a single match clause (i.e. conjunctive patterns); e.g:
MATCH (a)-(b)-(c), (b)-(f)

Input: a property graph
Output: a table
Cypher patterns

Node patterns

MATCH (node), (node:Node), (:Node), (node {type:"NODE"})

Relationship patterns

MATCH ()-->(), ()<--(), ()--() // Single relationship
MATCH ()-[edge]->(), (a)-[edge]->(b) // With binding
MATCH ()-[::RELATES]->() // With specific relationship type
MATCH ()-[edge {score:5}]->() // With property predicate
MATCH ()-[r:LIKES|:EATS]->() // Union of relationship types
MATCH ()-[r:LIKES|:EATS {age: 1}]->() // Union with property predicate
(applies to all relationship types specified)
Cypher patterns

Variable-length relationship patterns

MATCH (me)-[:FRIEND*]-(foaf)  // Traverse 1 or more FRIEND relationships
MATCH (me)-[:FRIEND*2..4]-(foaf)  // Traverse 2 to 4 FRIEND relationships
MATCH (me)-[:FRIEND*0..]-(foaf)  // Traverse 0 or more FRIEND relationships
MATCH (me)-[:FRIEND*2]-(foaf)  // Traverse 2 FRIEND relationships
MATCH (me)-[:LIKES|HATES*]-(foaf)  // Traverse union of LIKES and HATES 1 or more times

// Path binding returns all paths (p)
MATCH p = (a)-[:ONE]-()-[:TWO]-()-[:THREE]-()
// Each path is a list containing the constituent nodes and relationships, in order
RETURN p

// Variation: return all constituent nodes of the path
RETURN nodes(p)

// Variation: return all constituent relationships of the path
RETURN relationships(p)
Cypher: linear composition and aggregation

1: MATCH (me:Person {name: $name})-[[:FRIEND]]-(friend)
2: WITH me, count(friend) AS friends
3: MATCH (me)-[:ENEMY]-(enemy)
4: RETURN friends, count(enemy) AS enemies

**WITH** provides a *horizon*, allowing a query to be subdivided:
- Further matching can be done after a set of updates
- Expressions can be evaluated, along with aggregations
- Essentially acts like the pipe operator in Unix

**Linear composition**
- Query processing begins at the top and progresses linearly to the end
- Each clause is a function taking in a table $T$ (*line 1*) and returning a table $T'$
- $T'$ then acts as a driving table to the next clause (*line 3*)
Example query: epidemic!

Assume a graph G containing doctors who have potentially been infected with a virus....
The following Cypher query returns the name of each doctor in G who has perhaps been exposed to some source of a viral infection, the number of exposures, and the number of people known (both directly and indirectly) to their colleagues:

1: **MATCH** (d:Doctor)
2: **OPTIONAL MATCH** (d)-[:EXPOSED_TO]->(v:ViralInfection)
3: **WITH** d, count(v) **AS** exposures
4: **MATCH** (d)-[:WORKED_WITH]->(colleague:Person)
5: **OPTIONAL MATCH** (colleague)<-[[:KNOWS*]]-(p:Person)
6: **RETURN** d.name, exposures, count(DISTINCT p) **AS** thirdPartyCount
Example query

1: MATCH (d:Doctor)
2: OPTIONAL MATCH (d)-[:EXPOSED_TO]->(v:ViralInfection)

Matches all :Doctors, along with whether or not they have been :EXPOSED_TO a :ViralInfection

**OPTIONAL MATCH** analogous to outer join in SQL

- Produces rows provided entire pattern is found
- If no matches, a single row is produced in which the binding for v is null

<table>
<thead>
<tr>
<th>d</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>SourceX</td>
</tr>
<tr>
<td>Sue</td>
<td>PatientY</td>
</tr>
<tr>
<td>Alice</td>
<td>SourceX</td>
</tr>
<tr>
<td>Bob</td>
<td>null</td>
</tr>
</tbody>
</table>

Although we show the *name* property (for ease of exposition), it is actually the *node* that gets bound
Example query

3: **WITH** d, count(v) **AS** exposures

**WITH** projects a subset of the variables in scope - *d* - and their bindings onwards (to 4).

**WITH** also computes an aggregation:
- *d* is used as the grouping key implicitly (as it is not aggregated) for `count()`
- All non-null values of *v* are counted for each unique binding of *d*
- Aliased as *exposures*

The variable *v* is no longer in scope after 3

<table>
<thead>
<tr>
<th>d</th>
<th>exposures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>2</td>
</tr>
<tr>
<td>Alice</td>
<td>1</td>
</tr>
<tr>
<td>Bob</td>
<td>0</td>
</tr>
</tbody>
</table>
Example query

4: **MATCH (d)-[:WORKED_WITH]->(colleague:Person)**

Uses as driving table the binding table from 3

Finds all the colleagues (:Person) who have :WORKED_WITH our doctors

<table>
<thead>
<tr>
<th>d</th>
<th>exposures</th>
<th>colleague</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>2</td>
<td>Chad</td>
</tr>
<tr>
<td>Sue</td>
<td>2</td>
<td>Carol</td>
</tr>
<tr>
<td>Bob</td>
<td>0</td>
<td>Sally</td>
</tr>
</tbody>
</table>
Example query

5: **OPTIONAL MATCH** (colleague)<-[:KNOWS*]-(p:Person)

Finds all the people (:Person) who :KNOW our doctors’ colleagues (only in the one direction), both directly and indirectly (using :KNOWS* so that one or more relationships are traversed)

<table>
<thead>
<tr>
<th>d</th>
<th>exposures</th>
<th>colleague</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>2</td>
<td>Chad</td>
<td>Carol</td>
</tr>
<tr>
<td>Sue</td>
<td>2</td>
<td>Carol</td>
<td>null</td>
</tr>
<tr>
<td>Bob</td>
<td>0</td>
<td>Sally</td>
<td>Will</td>
</tr>
<tr>
<td>Bob</td>
<td>0</td>
<td>Sally</td>
<td>Chad</td>
</tr>
<tr>
<td>Bob</td>
<td>0</td>
<td>Sally</td>
<td>Carol*</td>
</tr>
<tr>
<td>Bob</td>
<td>0</td>
<td>Sally</td>
<td>Carol*</td>
</tr>
</tbody>
</table>

*This is due to the :KNOWS* pattern: Carol is reachable from Sally via Chad and Will (Carol :KNOWS Will and Chad)
Example query results

1: MATCH (d:Doctor)
2: OPTIONAL MATCH (d)-[:EXPOSED_TO]-(v:ViralInfection)
3: WITH d, count(v) AS exposures
4: MATCH (d)-[:WORKED_WITH]-(colleague:Person)
5: OPTIONAL MATCH (colleague)<-[:KNOWS*]-(p:Person)
6: RETURN d.name, exposures, count(DISTINCT p) AS thirdPartyCount

+-------------------------------------------+
| d.name | exposures | thirdPartyCount       |
+-------------------------------------------+
| Bob    | 0         | 3 (Will, Chad, Carol) |
| Sue    | 2         | 1 (Carol)             |
+-------------------------------------------+
Other functionality

Aggregating functions

- `count()`, `max()`, `min()`, `avg()`, ...

Operators

- Mathematical, comparison, string-specific, boolean, list

Map projections

- Construct a map projection from nodes, relationships and properties

**CASE** expressions, functions (scalar, list, mathematical, string, UDF, procedures)
Formal semantics

The meaning of Cypher clauses is again functions that take tables to tables. Matching clauses are essentially pattern matching statements: they are of the form \texttt{OPTIONAL MATCH} pattern\_tuple \texttt{WHERE} expr. Both \texttt{OPTIONAL} and \texttt{WHERE} could be omitted. The key to their semantics is pattern matching, in particular \texttt{match(π, G, u)} described in Section 4 (see Equation (1), page 12).

The \texttt{MATCH} clause extends the set of field names of $T$ by adding to it field names that correspond to names occurring in the pattern but not in $u$. It also adds tuples to $T$, based on matches of the pattern that are found in graphs. \texttt{UNWIND} is another clause that expands the set fields, and \texttt{WITH} clauses can change the set of fields to any desired one. The \texttt{WHERE} subclause also defines a table-to-tables function that filters lines according to the evaluation of an expression; it is not a proper clause because of its interaction with \texttt{OPTIONAL MATCH} clauses.

Matching clause The semantics of \texttt{MATCH} clauses is defined below; the semantics of \texttt{WHERE} subclause is defined afterwards.

- $\mathbb{[MATCH \ π]}_G(T) = \bigcup\limits_{u \in T} \{u \cdot u' \mid u' \in \text{match}(\pi, G, u)\}$
Evolving Cypher through the openCypher project
openCypher...

...is a community effort to evolve Cypher, and make it the de-facto language for querying property graphs

**openCypher implementations**

<table>
<thead>
<tr>
<th>Industrial</th>
<th>Research</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAP HANA Graph</td>
<td>Gradoop (<em>Distributed Graph Analytics on Apache Flink</em>): U. of Leipzig</td>
</tr>
<tr>
<td>Redis Graph</td>
<td><em>ingraph</em> (<em>Incremental evaluation of Cypher queries</em>): U. of Budapest</td>
</tr>
<tr>
<td>Agens Graph</td>
<td><strong>Graphflow</strong> (<em>Supporting continuous queries and triggers</em>): U. of Waterloo</td>
</tr>
<tr>
<td>Neo4j</td>
<td></td>
</tr>
<tr>
<td>CAPS</td>
<td></td>
</tr>
<tr>
<td>Cypher for Gremlin</td>
<td></td>
</tr>
</tbody>
</table>
openCypher

openCypher Implementers Group (oCIG)

Evolve Cypher through an open process

Comprises vendors, researchers, implementers, interested parties

Regular meetings to discuss and agree upon new features

Consensus-based system
The Second openCypher Implementers Meeting
openCypher website

Blog
Cypher 9 Reference
New Features
Upcoming Meetings
Recordings and Slides
References (Links, Papers)
Artifacts
github.com/openCypher

Language Artifacts

Cypher 9 reference
ANTLR and EBNF Grammars
Formal Semantics (SIGMOD, to be published here)
Technology Compatibility Kit (TCK) - Cucumber test suite
Style Guide

Implementations & Code

openCypher for Apache Spark
openCypher for Gremlin
Open source frontend (part of Neo4j, to be published here)
**TCK (Technology Compliance Kit)**

**Scenario:** Optionally matching named paths

**Given** an empty graph

**And** having executed:

```
CREATE (a {name: 'A'}), (b {name: 'B'}), (c {name: 'C'})
CREATE (a)-[:X]->(b)
```

**When** executing query:

```
MATCH (a {name: 'A'}), (x)
WHERE x.name IN ['B', 'C']
OPTIONAL MATCH p = (a)-->(x)
RETURN x, p
```

**Then** the result should be:

<table>
<thead>
<tr>
<th>x</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>({name: 'B'})</td>
<td>&lt;({name: 'A'})-[:X]-&gt;({name: 'B'})&gt;</td>
</tr>
<tr>
<td>({name: 'C'})</td>
<td>null</td>
</tr>
</tbody>
</table>

**And** no side effects

**Background:**

**Given** any graph

**Scenario:** Creating a node

**When** executing query:

```
CREATE ()
```

**Then** the result should be empty

**And** the side effects should be:

| +nodes | 1 |

*based on slide by M. Rydberg*
Cypher extensions
Extension: Multiple graphs and query composition
Cypher today: single graph model
Cypher: multiple graphs model

Application Server

Client 1
Client 2
Client 3

Graph Database System
(e.g. a cluster)

Multiple Graphs
Why multiple graphs?

Combining and transforming graphs from **multiple sources**

Versioning, snapshotting, computing difference graphs

**Graph views** e.g. for access control

Shaping and **integrating** heterogeneous graph data
Classic Property Graph Model

- A single graph
- Labelled nodes with properties
- Typed, directed relationships with properties

Multiple Property Graphs Model

- Multiple graphs
- Labelled nodes with properties existing in a single graph
- Typed, directed relationships with properties
- The nodes of a relationship $R$ must be part of the same graph as $R$; i.e. each graph is independent
Query composition

The output of one query can be used as the input to another

Organize a query into multiple parts

Extract parts of a query to a view for re-use

Replace parts of a query without affecting other parts

Build complex workflows programmatically
Cypher query pipeline composition
Implications for Cypher

Pass both multiple graphs and tabular data into a query

Return both multiple graphs and tabular data from a query

Select which graph to query

Chain queries to form a query pipeline

Construct new graphs from existing graphs

Catalog management functions - creating a graph, copying a graph etc - are also under consideration
Matching graphs and constructing graphs

Duality:

**MATCH** turns graphs into matches for the pattern

**CONSTRUCT** turns matches for the pattern back into graphs
Linear composition: potential syntax

```
FROM GRAPH G_1
MATCH (a)
FROM GRAPH G_2
MATCH (b)
CONSTRUCT
  ON G_1, G_2
  CLONE a, b
  NEW (a)-[:NEW_REL]->(b)
RETURN GRAPH
```
Linear composition: Cypher vs. SQL

MATCH (a:B)
WHERE a.foo % 42 = 0
WITH a.id as id,
    a.foo as foo,
    a.bar as bar
WHERE bar > 3
WITH id, foo, count(*) as X
WHERE X > 100
RETURN foo + " " + id

SELECT foo + " " + id
FROM (SELECT id, foo, count(*) as X FROM (SELECT id, foo, bar FROM B WHERE foo % 42 = 0) AS T1 WHERE bar > 3 GROUP BY id, foo) as T2
WHERE X > 100

Cypher appends
SQL nests
Tree composition using subqueries

**Nested**
- Run any complete read-only Cypher query
- Incoming variables remain in scope: correlated subquery
- Arbitrary depth

**Existential**
- Returns true if at least one match found; false otherwise

**Scalar**
- Result is a single value in a single row

**List**
- Result is the list formed by collecting all the values of all rows (single value per row)
Subqueries: tentative syntax proposal

<outer pre-query>
CALL {
  <inner reading subquery>
  ...
  // produce additional rows
  RETURN ...
}
<outer post-query>
Multiple graphs and named queries

Multiple graphs introduce **named graph references**, which represent any of the following:

- Externally-located graphs (e.g., from a catalog)
- Graphs created by the query
- Graphs created by a previous query in a composition of queries

**Named queries** are also introduced

They simplify the creation of libraries of re-usable queries which can be composed in different query pipelines

They form the basis for offering graph views
Extension: Complex path patterns
Complex path patterns

Regular path queries (RPQs)

X, (likes.hates)*(eats|drinks)+, Y

Find a path whose edge labels conform to the regular expression, starting at node X and ending at node Y

(X and Y are node bindings)

I. F. Cruz, A. O. Mendelzon, and P. T. Wood

A graphical query language supporting recursion


Plenty of research in this area since 1987!

SPARQL 1.1 has support for RPQs: “property paths”
RPQs - aka ‘Path Pattern Queries’ - in Cypher

Find complex connections

Customers are beginning to ask for the capability to express “nested patterns”; in particular: \((a \cdot b)^*\)

Property graph data model:

- **Properties** need to be considered
- **Node labels** need to be considered

Specifying a cost for paths (ordering and comparing)

CIP: [https://github.com/thobe/openCypher/blob/rpq/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc](https://github.com/thobe/openCypher/blob/rpq/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc)
Academic research: Path Patterns

Functionality of RPQs
  Relationship types

Using **GXPath** as inspiration
  Node tests
  Relationship tests

Not considering unreachable (via a given path) pairs of nodes: **intractable**

Context Free Languages and Regular Expressions with Binding
  Corresponding constructs in Cypher’s Path Pattern Queries

L. Libkin, W. Martens, and D. Vrgoč
*Querying Graphs with Data*
Predicates for Path Patterns

Relationship type: 

Node label: 

Relationship properties: 

Node properties: 

Provisional syntax

https://www.opencypher.org
Composition of Path Patterns

Sequence / Concatenation:

\( (\cdot)/ \alpha \beta /\cdot(\cdot) \)

Alternation / Disjunction:

\( (\cdot)/ \alpha \mid \beta /\cdot(\cdot) \)

Transitive closure:

- 0 or more
  \( (\cdot)/ \alpha^* /\cdot(\cdot) \)
- 1 or more
  \( (\cdot)/ \alpha^+ /\cdot(\cdot) \)
- \(n\) or more
  \( (\cdot)/ \alpha^n.. /\cdot(\cdot) \)
- At least \(n\), at most \(m\)
  \( (\cdot)/ \alpha^n..m /\cdot(\cdot) \)

Overriding direction for sub-pattern:

- Left to right direction
  \( (\cdot)/ \alpha > /\cdot(\cdot) \)
- Right to left direction
  \( (\cdot)/ < \alpha /\cdot(\cdot) \)
- Any direction
  \( (\cdot)/ < \alpha > /\cdot(\cdot) \)
Path Pattern: example

**PATH PATTERN**

older_friends = (a)-[::FRIEND]-(b) WHERE b.age > a.age

**MATCH**
p=(me)-/~older_friends+/-(you)
WHERE me.name = $myName AND you.name = $yourName
RETURN p AS friendship
**Nested Path Patterns: example**

**PATH PATTERN**

`older_friends = (a)-[:FRIEND]-(b) WHERE b.age > a.age`

**PATH PATTERN**

`same_city = (a)-[:LIVES_IN]->(:City)<-[[:LIVES_IN]]-(b)`

**PATH PATTERN**

`older_friends_in_same_city = (a)-/~older_friends/~-(b)`

**WHERE EXISTS**

`{ (a)-/~same_city/~-(b) }`
Cost function for cheapest path search

PATH PATTERN road = (a)-[r:ROAD_SEGMENT]-(b) COST r.length

MATCH route = (start)-/^road*/-(end)

WHERE start.location = $currentLocation
AND end.name = $destination

RETURN route

ORDER BY cost(route) ASC LIMIT 3
Extension: Configurable pattern-matching semantics
“Cyphermorphism”

Pattern matching today uses relationship isomorphism (no repeated relationships)

MATCH (p:Person {name: Jack})-[r1:FRIEND]-()-[r2:FRIEND]-(friend_of_a_friend)
RETURN friend_of_a_friend.name AS fofName

<table>
<thead>
<tr>
<th>fofName</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Tom”</td>
</tr>
</tbody>
</table>

r1 and r2 may not be bound to the same relationship within the same pattern

Usefulness proven in practice over multiple industrial verticals: we have not seen any worst-case examples

Rationale was to avoid potentially returning infinite results for varlength patterns when matching graphs containing cycles (this would have been different if we were just checking for the existence of a path)
Overriding Cyphermorphism today

MATCH (p:Person {name: 'Jack'})-[r1:FRIEND]-(friend)
MATCH (friend)-[r2:FRIEND]-(friend_of_a_friend)
RETURN friend_of_a_friend.name AS fofName

<table>
<thead>
<tr>
<th>fofName</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Tom”</td>
</tr>
<tr>
<td>“Jack”</td>
</tr>
</tbody>
</table>

r1 and r2 may now be bound to the same relationship as they appear in two distinct patterns.
All the *morphisms

Node isomorphism:

- No node occurs in a path more than once
- Most restrictive

Relationship isomorphism (Cyphermorphism):

- No relationship occurs in a path more than once
- Proven in practice

Homomorphism:

- A path can contain the same nodes and relationships more than once
- Most efficient for some RPQs
- Least restrictive
Configurable pattern matching

Allow all three forms of pattern matching

Recognition that *all forms are valid* in differing scenarios

The user can configure which semantics they wish to use at a query level

Configurable **pattern morphism** specifies the morphism type of the match

- **HOMOMORPHIC** Homomorphism
- **UNIQUE RELS** Relationship isomorphism
- **UNIQUE NODES** Node isomorphism

Illustrative syntax only!
Pattern quantifiers and length restrictions

Configurable **pattern quantifiers** controlling how many matches are returned

- **ANY**  At most one match (existence checking)
- **EACH** All matches

Configurable **pattern length restrictions** limits the length and nature of matches

- **SHORTEST**  Consider only shortest path (determined by length of path)
- **CHEAPEST**  Consider only cheapest path (determined by **COST** function in Path Pattern)
- **UNRESTRICTED**  Consider all possible paths

Example: **MATCH EACH SHORTEST HOMOMORPHISM <pattern>**
Recommended defaults

If a pattern is given no pattern morphism, pattern length restriction and pattern quantifier:

- EACH UNRESTRICTED UNIQUE RELS (the behaviour today)

If a pattern is SHORTEST or CHEAPEST:

- No pattern morphism specified: HOMOMORPHISM
- No pattern quantifier: ANY
(:Cypher)-[:FOR]->(:Apache:Spark™)
About me

Software Engineer at Neo4j

Working on Cypher for Apache Spark

PhD student at the University of Leipzig (Database chair, Prof. Rahm)

Research on distributed graph analytics (Graph Grouping, Pattern Matching, ...)

Developer and maintainer of the Gradoop project (www.gradoop.com)
About the CAPS team

Martin Junghanns

Max Kießling

Mats Rydberg

Philip Stutz
Motivation
Why Cypher for Big Data?

Cypher originally conceived in the context of OLTP workloads at Neo4j.

However, many Neo4j customers have data lakes and use Big Data tools for:

- Data integration (ETL)
- Large-scale analytical processing (OLAP)
Why Cypher for Big Data?

Today's Big Data applications ...

- Collect data from user interactions at website
- Combine with other data from various departments (billing, marketing, ...)
- Analyze to optimize supply chains, detect fraud, ...
- Use a common framework: Apache Spark
How can Cypher help Big Data analytics?

● Data integration
  ○ Use and integrate multiple, large-scale data sets
  ○ Retain and reuse intermediate results
  ○ Shape and handle heterogeneous data

● Complex Analytical Processing
  ○ Compose complex workflows from building blocks
  ○ Combine graph querying with machine learning, graph algorithms, domain specific business logic etc.
Introducing CAPS

Scala library for the execution of Cypher on Apache Spark

- Execute Cypher queries on multiple large, distributed graphs
- Integrate Cypher into a Spark analytical pipeline
- Integrate multiple data sources (Neo4j, Apache HDFS, Local FS, ...)
- Handle heterogeneous data
- Compose Cypher queries
What is CAPS?

- Developed by Neo4j, donated to the openCypher community
- Alpha release of source code under Apache 2.0 license on GitHub: [github.com/openCypher/cypher-for-apache-spark](https://github.com/openCypher/cypher-for-apache-spark)
- Planned GA Release: end of May, 2018 (for Spark + AI Summit)
  - Supported data sources: Neo4j, HDFS and Hive via SQL
  - 2018 / 2019: support for additional data sources
    - Relational DBMS (Oracle, PostgreSQL etc.)
    - Cypher-for-Gremlin and RDF database systems
- Technical innovations:
  - Architecture for executing Cypher on a Big Data analytics system
  - Support for composable queries and multiple graphs
(:Cypher)-[:FOR]->(Apache:Spark™)

Apache Spark
What is Apache Spark?

- Open source project https://spark.apache.org/ with ~1200 contributors
- Distributed, batch-oriented, in-memory dataflow system
- Efficiently supports non-linear dataflows
  - Enabled through caching of intermediate results in memory
  - Allows for iterative machine learning and graph algorithms
- Application logic is expressed using higher order functions on distributed datasets
  - map, flatMap, reduce, ...
- Already includes many libraries / abstractions for domain specific analytics

What is Apache Spark SQL?

- Introduces a relational abstraction on top of Spark Core API
- Central concept: DataFrame
  - Basically a distributed relation, i.e., a distributed table
  - Supports structured and semi-structured data (e.g. from/to RDBMS, CSV, Parquet, ...)
  - Manipulation via Scala DSL for relational algebra (or SQL using temporary views)

```
val df = spark.read.csv("examples/src/main/resources/people.csv")

df.select("name").show()
+--------+
<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Petra</td>
</tr>
<tr>
<td>Martin</td>
</tr>
<tr>
<td>--------</td>
</tr>
</tbody>
</table>

df.filter("$team" == "CAPS").select("name").show()
+--------+
<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Martin</td>
</tr>
<tr>
<td>---------</td>
</tr>
</tbody>
</table>

df.groupBy("team").count().show()
+------+-------+
| team | count |
+------+-------|
| CAPS | 1     |
| CLG  | 1     |
+------+-------+

What is Apache Spark SQL?

- Benefits of using Spark SQL
  - Declarative queries are less error-prone than manual procedural programs
  - Declarative queries enable richer automatic optimization
  - Known schema allows for data type specific optimizations (e.g. column compression)

- Spark Catalyst
  - Extensible, relational query optimization framework contained in Spark SQL
  - Rule-based query optimizer (Apache Spark 2.3.0 also includes a cost-based optimizer)

- Spark SQL is a good foundation for a query translation engine like CAPS
  - Tradeoffs: Schema-optional vs Schema-fixed, different type systems, etc.
(:Cypher)-[:FOR]->(Apache:Spark™)

Technical Architecture
High Level Architecture

MATCH (n:Person)-[:LOVES]->(s:System)
WHERE n.name = 'Alice'
RETURN n.name, s.name

- Shared with Neo4j database system
- Parsing, Rewriting, Normalization
- Semantic Analysis (Scoping, Typing, etc.)
- Data Import and Export
- Schema and Type handling
- Query translation to DataFrame operations
- Rule-based query optimization
- Distributed execution
## Challenge: Graph engine vs. Relational engine

<table>
<thead>
<tr>
<th></th>
<th>Neo4j</th>
<th>Spark SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Schema</strong></td>
<td>Schema optional</td>
<td>Fixed Schema</td>
</tr>
<tr>
<td><strong>Graph Format</strong></td>
<td>Native (i.e. optimized for graph ops)</td>
<td>DataFrame (i.e. tables)</td>
</tr>
<tr>
<td><strong>Query operators</strong></td>
<td>Native (e.g. Expand, VarExpand)</td>
<td>Relational operators</td>
</tr>
<tr>
<td><strong>Data types</strong></td>
<td>Cypher type system</td>
<td>Spark SQL type system</td>
</tr>
</tbody>
</table>
Challenge: Schema

- Property Graphs are schema-optional
- DataFrames require a fixed schema upon creation

Node labels combinations:
- :Employee:Person
  - { name : Alice }  
- :Person
  - { name : Bob, yob : 1984 }
- :System
  - { title : Spark }
- :System
  - { title : Neo4j }

Relationship types:
- :KNOWS
  - { since : 2017 }
- :LOVES

Implied Labels:
- :Employee -> :Person

- Schema computation depends on data source
  - e.g. explicitly defined (e.g. for HDFS data source)
  - e.g. implicitly inferred (e.g. for Neo4j data source)

Property Graphs are schema-optional
DataFrames require a fixed schema upon creation

Node labels combinations:
- :Person:Employee
  - name: CTString
- :Person
  - name: CTString
  - yob: CTInteger.nullable
- :System
  - title: CTString

Relationship types:
- :KNOWS
  - since: CTInteger
- :LOVES

Implied Labels:
- :Employee -> :Person
Challenge: Schema

- Requires type mapping from Cypher types to Spark types

<table>
<thead>
<tr>
<th>CypherType</th>
<th>Spark DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTNull</td>
<td>CTVoid</td>
</tr>
<tr>
<td>CTInteger</td>
<td></td>
</tr>
<tr>
<td>CTString</td>
<td></td>
</tr>
<tr>
<td>CTBoolean</td>
<td></td>
</tr>
<tr>
<td>CTFloat</td>
<td></td>
</tr>
<tr>
<td>CTList(CypherType)</td>
<td></td>
</tr>
</tbody>
</table>

- CAPS includes a type system that defines a type hierarchy (necessary for type joining), comparison and orderability semantics
Challenge: Graph representation

Logical view

Physical view (DataFrame)

<table>
<thead>
<tr>
<th>NodeTable(Person)</th>
<th>NodeTable(System)</th>
<th>RelTable(KNOWS)</th>
<th>RelTable(LOVES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>n.Employee</td>
<td>n.name</td>
<td>n.yob</td>
</tr>
<tr>
<td>0</td>
<td>true</td>
<td>Alice</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>false</td>
<td>Bob</td>
<td>1984</td>
</tr>
</tbody>
</table>
Challenge: Query translation

MATCH (n:Person)-[:LOVES]->(s:System)
WHERE n.name = 'Alice'
RETURN n.name, s.name

Logical view

Physical view (DataFrame operations)
CAPS Architecture

openCypher Frontend

Cypher for Apache Spark

Spark SQL

Spark Core

Intermediate Language

Backend Agnostic Query Representation

Conversion of expressions

Typing of expressions

Logical Planning

Translation into Logical Operators

Basic Logical Optimization

Flat Planning

Column layout computation for intermediate results

Physical Planning

Translation into Spark SQL DataFrame operations
MATCH (n:Person)-[:KNOWS]->(s:System)
WHERE n.name = 'Alice'
RETURN n.name, s.title
Intermediate Language

**MATCH** (n:Person)-[:KNOWS]->(s:System)
WHERE n.name = ‘Alice’
RETURN n.name, s.title

- TableResultBlock(OrderedFields(List(n.name :: STRING, s.title :: STRING)), ...)
- ProjectBlock(Fields(Map(n.name :: STRING -> n.name :: STRING, s.title :: STRING -> s.title :: STRING)), List(), ...)
- MatchBlock(Pattern(
  Set(n :: NODE, s :: NODE, UNNAMED18 :: :KNOWS RELATIONSHIP),
  Map( UNNAMED18 :: :KNOWS RELATIONSHIP -> DirectedRelationship(Endpoints(n :: NODE, s :: NODE))), Map()),
  List(n:Person :: BOOLEAN, s:System :: BOOLEAN, n.name :: STRING = $ AUTOSTRING0 :: STRING), ...)
- SourceBlock(...)
Logical Plan

MATCH (n:Person)-[:KNOWS]->(s:System)
WHERE n.name = 'Alice'
RETURN n.name, s.title

|-Select(List(n.name :: STRING, s.title :: STRING), Set(), ...)
  |-Project(s.title :: STRING, Some(s.title :: STRING), ...)
  |-Project(n.name :: STRING, Some(n.name :: STRING), ...)
    |-Filter(n.name :: STRING = $ AUTOSTRING0 :: STRING, ...)
    |-Project(n.name :: STRING, None, ...)
    |-Filter(s:System :: BOOLEAN, ...)
      |-Filter(n:Person :: BOOLEAN, ...)
        |-Expand(n :: NODE, UNNAMED18 :: :KNOWS RELATIONSHIP, s :: NODE, Directed, ...)
        |-NodeScan(n :: NODE, ...)
          |-Start(LogicalCatalogGraph(qualifiedGraphName = session.4c11e144), ...)
        |-NodeScan(s :: NODE, ...)
          |-Start(LogicalCatalogGraph(qualifiedGraphName = session.4c11e144), ...)

- Convert Intermediate Language Blocks into Logical Query Operators
Optimized Logical Plan

```cypher
MATCH (n:Person)-[:KNOWS]->(s:System)
WHERE n.name = 'Alice'
RETURN n.name, s.title
```

- Select(List(n.name :: STRING, s.title:: STRING), Set(), ...)
  - Project(s.title :: STRING, Some(s.title :: STRING), ...)
    - Project(n.name :: STRING, Some(n.name :: STRING), ...)
      - Filter(n.name :: STRING = $AUTOSTRING0 :: STRING, ...)
        - Project(n.name :: STRING, None, ...)
          - Expand(n :: NODE, UNNAMED18 :: :KNOWS RELATIONSHIP, s :: NODE, Directed, ...)
            - NodeScan(n :: :Person NODE, ...)
            - Start(LogicalCatalogGraph(qualifiedGraphName = session.4c11e144), ...)
        - NodeScan(s :: :System NODE, ...)
        - Start(LogicalCatalogGraph(qualifiedGraphName = session.4c11e144), ...)

- Apply basic optimizations to a Logical Query plan (e.g. label predicate pushdown)
Flat plan

```
MATCH (n:Person)-[:KNOWS]->(s:System)
WHERE n.name = 'Alice'
RETURN n.name, s.title
```

- Select(List(n.name :: STRING, s.title :: STRING), Set(), RecordHeader with 2 slots)
  - Alias(s.title :: STRING, s.title :: STRING, RecordHeader with 11 slots)
    - Alias(n.name :: STRING, n.name :: STRING, RecordHeader with 11 slots)
      - Filter(n.name :: STRING = $ AUTOSTRING0 :: STRING, RecordHeader with 11 slots)
      - Expand(n :: NODE, UNNAMED18 :: :KNOWS_RELATIONSHIP, Directed, s :: NODE, RecordHeader with 11 slots, ...)
    - NodeScan(n :: :Person NODE, RecordHeader with 3 slots)
  - Start(LogicalCatalogGraph(qualifiedGraphName = session.4c11e144), Set())
  - NodeScan(s :: :System NODE, RecordHeader with 3 slots)
  - Start(LogicalCatalogGraph(qualifiedGraphName = session.4c11e144), Set())

- Compute the column header for the underlying table abstraction (i.e. the SparkSQL DataFrame)
Record Header

- Describes the output table of a flat operator
- Maps query expression to DataFrame/Table column names
- Used to access the right column when evaluating expression during physical execution

NodeScan(n :: :Person NODE, RecordHeader with 3 slots)

<table>
<thead>
<tr>
<th>Record Slot</th>
<th>Column Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(id, SlotContent(Expression))</td>
<td></td>
</tr>
<tr>
<td>0, OpaqueField(n :: :Person CTNODE)</td>
<td>n</td>
</tr>
<tr>
<td>1, ProjectedExpr(n:Person :: CTBOOLEAN)</td>
<td>____n:Person</td>
</tr>
<tr>
<td>2, ProjectedExpr(n.name :: CTSTRING)</td>
<td>____n_dot_nameSTRING</td>
</tr>
</tbody>
</table>
Physical plan

```
MATCH (n:Person)-[:KNOWS]->(s:System)
WHERE n.name = 'Alice'
RETURN n.name, s.title
```

- SelectFields(List(n.name :: STRING, s.title :: STRING), RecordHeader with 2 slots)
  - Alias(s.title :: STRING, s.title :: STRING, RecordHeader with 11 slots)
  - Alias(n.name :: STRING, n.name :: STRING, RecordHeader with 11 slots)
  - Filter(n.name :: STRING = $AUTOSTRING0 :: STRING, RecordHeader with 11 slots)
    - ExpandSource(n :: NODE, UNNAMED18 :: :KNOWS RELATIONSHIP, s :: NODE, RecordHeader with 11 slots, false)
    - Scan(n :: :Person NODE, RecordHeader with 3 slots)
    - Start(Some(CAPSRecords.unit), Some(LogicalCatalogGraph(qualifiedGraphName = session.4c86d2c7)))
    - Scan(UNNAMED18 :: :KNOWS RELATIONSHIP, RecordHeader with 5 slots)
    - Start(None, Some(LogicalCatalogGraph(qualifiedGraphName = session.4c86d2c7)))
    - Scan(s :: :System NODE, RecordHeader with 3 slots)
    - Start(Some(CAPSRecords.unit), Some(LogicalCatalogGraph(qualifiedGraphName = session.4c86d2c7)))

- Physical operators (e.g. Scan, Expand, Filter) translate into Spark SQL DataFrame operations
Physical Operator Implementation

```scala
// simplified implementation
final case class ExpandSource(source: Var, rel: Var, target: Var, targetHeader: RecordHeader) {
  def execute(left: CAPSPhysicalResult, rels: CAPSPhysicalResult, right: CAPSPhysicalResult): CAPSPhysicalResult = {

    // get DataFrames and RecordHeaders
    val leftDf     = left.records.data;
    val relsDf     = rels.records.data;
    val rightDf     = right.records.data
    val leftHeader = left.records.header;
    val relsHeader = rels.records.header;
    val rightHeader = right.records.header

    // extract Join columns
    val sourceColumn      = leftDf.col(columnName(leftHeader.slotFor(source)))
    val sourceNodeColumn  = relsDf.col(columnName(relsHeader.sourceNodeSlot(rel)))
    val targetNodeColumn  = relsDf.col(columnName(relsHeader.targetNodeSlot(rel)))
    val targetColumn      = rightDf.col(columnName(rightHeader.slotFor(target)))

    // join DataFrames
    val expandDf = leftDf
                              .join(relsDf,  sourceColumn === sourceNodeColumn)
                              .join(rightDf, targetNodeColumn === targetColumn)

    // create result
    val expandRecords = CAPSRecords.verifyAndCreate(header, expandDf)
    CAPSPhysicalResult(expandRecords, left.graph)
  }
}
```

Challenge: Representing results

Logical view

Physical view (DataFrame)

<table>
<thead>
<tr>
<th>CypherResult</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n.name</td>
<td>s.title</td>
</tr>
<tr>
<td>Alice</td>
<td>Spark</td>
</tr>
<tr>
<td>Alice</td>
<td>Neo4j</td>
</tr>
</tbody>
</table>
(:Cypher)-[:FOR]->(Apache:Spark™)

Reusable Architecture
Reusable architecture

- openCypher Frontend
- Intermediate Language
  - Logical Planning
  - Flat Planning
  - Physical Planning
- Spark SQL
- Spark Core

- okapi-api
  - Property Graph API
  - Type System
  - Data Source API
- okapi-ir
  - Intermediate Language
- okapi-logical
  - Logical Planning
- okapi-relational
  - Flat Planning
  - Physical Planning

Abstraction
Backend specific Implementation

- spark-cypher
  - Physical Operators
- flink-cypher
  - Physical Operators
- mem-cypher
  - Physical Operators

OKAPI Pipeline
(:Cypher)-[:FOR]->(Apache:Spark™)

Programming API
CAPS API: CAPSSession

- Programmatic, high-level API (similar to Spark’s DataFrame API)
- Central entry point: `CAPSSession`

```scala
1: implicit val session: CAPSSession = CAPSSession.local()
2: val socialNetwork = session.readFrom(SocialNetworkData.persons, SocialNetworkData.friendships)
3: val result = socialNetwork.cypher("MATCH (n) RETURN n.name, n.age")
4: result.show
```

+---------------------------------------------+
| n.name               | n.age     |
+---------------------------------------------+
| 'Alice'              | 10        |
| 'Bob'                | 20        |
| 'Carol'              | 15        |
+---------------------------------------------+
(3 rows)

CAPS API: Property Graph Data Source

- OKAPI-API provides interface for Data Source implementations
- Data Sources can manage multiple graphs
- Data Sources are registered within a Catalog (i.e. the Session) using a Namespace
- Graphs are addressed using their Qualified Graph Name (i.e. Namespace and Graph Name)

```scala
1: val neo4jSource = new Neo4jPropertyGraphDataSource(neo4jConfig)
2: session.registerSource(Namespace("neo4j"), neo4jSource)
3: val friendOfAFriend = session.cypher(""
    |FROM GRAPH neo4j.graph
    |MATCH (n:Person)-[:KNOWS]->()-[:KNOWS]->(m:Person)
    |WHERE NOT (n)-[:KNOWS]->(m)
    |RETURN n, m"".stripMargin)
```

CAPS API: Data Integration

- Refer to multiple graphs within the query and combine them using `CONSTRUCT`

```
1: val integrationGraph = session.cypher(""
|FROM GRAPH neo4j.graph
|MATCH (p:Person)
|FROM GRAPH csv.products
|MATCH (c:Customer)
|WHERE p.name = c.name
|CONSTRUCT ON neo4j.graph, csv.products
|  CLONE p
|  CLONE c
|  NEW (p)-[x:IS]->(c)
|RETURN GRAPH
""".stripMargin).graph.get

2: val recommendations = integrationGraph.cypher(""
|MATCH (person:Person)-[:FRIEND_OF]-(friend:Person),
|  (friend)-[:IS]->(customer:Customer),
|  (customer)-[:BOUGHT]->(product:Product)
|RETURN person.name AS for,
|  collect(DISTINCT product.title) AS recs"
"
3: recommendations.show
```
CAPS API: Spark SQL interaction

- Cypher Results can be used as input table for Spark SQL
- SQL results can be used as driving table for a Cypher Query

```scala
1: val result = socialNetwork.cypher(
    |MATCH (p:Person)
    |RETURN p.age AS age, p.name AS name"
   ).stripMargin
   // Register the result as a table called people
2: result.getRecords.register("people")
   // Query the registered table using SQL
3: val sqlResults = session.sql("SELECT age, name FROM people")
   // Use the results as driving table for another Cypher query
4: val result2 = session.cypher(""
   |FROM GRAPH csv.products
   |MATCH (c:Customer {name: name})-->(p:Product)
   |RETURN c.name, age, p.title"
   ).stripMargin, drivingTable = Some(sqlResults))
```

(:Cypher)-[:FOR]-->([ApACHE:SpArk™])

Demo
Target specific customers in selected metropolitan areas as part of a marketing campaign

Combine multi-region social network data with product data to derive recommendations

Social network is partitioned by region (SN_NA, SN_EU) and stored in Neo4j

Product data is stored in HDFS using a CAPS-specific CSV format
1. Load data from the corresponding data sources (i.e. Neo4j and HDFS)
2. Extract metropolitan subgraphs from Social Networks (e.g. people from NY / SFO for SN_NA)
3. Merge Social Network data with Product data using identifying properties (e.g. email)
4. Compute recommendations based on friends’ interests and bought products
(:Thank)-[::]->(:You)

github.com/openCypher/cypher-for-apache-spark
(Cypher)-[:FOR]-(Apache:Spark™)

Demo
Getting involved

Please follow news at opencypher.org and @opencypher on twitter

There's a great slack channel for implementers

Join the Google group at https://groups.google.com/forum/#!forum/opencypher

Language change request issues (CIRs) and full proposals (CIPs)

Own ideas? Talk to us! Or create a Pull Request at https://github.com/opencypher/openCypher
Thank you!

If you have a research topic, please come and speak to us!

petra.selmer@neo4j.com; martin.junghanns@neo4j.com