Graph databases and graph querying

Advances in Data Management, 2019

Dr. Petra Selmer
Query languages standards & research group, Neo4j
About me

Member of the Query Languages Standards & Research Group at Neo4j

Collaborations with academic partners in graph querying

Design new features for graph querying

Standardisation efforts within ISO: GQL (Graph Query Language)

Manage the openCypher project

Previously: engineer at Neo4j

Work on the Cypher Features Team

PhD in flexible querying of graph-structured data (Birkbeck, University of London)
The property graph data model
The Cypher query language
Introducing Graph Query Language (GQL)
GQL Features
  - Graph pattern matching
  - Type system
  - Expressions
  - Schema and catalog
  - Modifying and projecting graphs
  - Query composition and views
  - Other work
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](https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc)
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))

Edge
- 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))

Edge

- 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 edges
- Represents the data: e.g. name, age, weight etc
- String key; typed value (string, number, bool, list)
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
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
Data centre dependency network

Nodes model applications, servers, racks, etc

Edges model how these entities are connected

Impact analysis
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

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

MATCH (:Person { name:"Dan"}) -[LOVES]-> (whom) RETURN whom
Cypher: nodes

() or (n)

Surround with parentheses

Use an alias n to refer to our node later in the query

(n:Label)

Specify a Label, starting with a colon:

Used to group nodes by roles or types (similar to tags)

(n:Label {prop: 'value'})

Nodes can have properties
Cypher: edges / relationships

--> or -[:TYPE]-->

Wrapped in hyphens and square brackets

A relationship type starts with a colon :

<>

Specify the direction of the relationships

-[:KNOWS {since: 2010}]-->

Relationships can have properties
Cypher: patterns

Used to query data

(n:Label {prop: 'value'})-[:TYPE]->(m:Label)
Find Alice who knows Bob

In other words:

find Person with the name ‘Alice’

who KNOWS

a Person with the name ‘Bob’

(p1:Person {name: ‘Alice’})-[:KNOWS]->(p2:Person {name: ‘Bob’})
// 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
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**)

Parameters: $\$\text{param}$

Aggregation (grouped by ‘me’)
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

This binding table is now the driving table for the **MATCH** in 4

<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: \textbf{MATCH} (d)-[:\text{WORKED\_WITH}]->(colleague:Person)

Uses as driving table the binding table from 3

Finds all the colleagues :\texttt{Person} who have :\texttt{WORKED\_WITH} our doctors

<table>
<thead>
<tr>
<th></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>
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>
</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

<table>
<thead>
<tr>
<th>d.name</th>
<th>exposures</th>
<th>thirdPartyCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>0</td>
<td>3 (Will, Chad, Carol)</td>
</tr>
<tr>
<td>Sue</td>
<td>2</td>
<td>1 (Carol)</td>
</tr>
</tbody>
</table>
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)
Introducing Graph Query Language (GQL)
Many implementations

Amazon Neptune, Oracle PGX, Neo4j Server, SAP HANA Graph, AgensGraph (over PostgreSQL), Azure CosmosDB, Redis Graph, SQL Server 2017 Graph, Cypher for Apache Spark, Cypher for Gremlin, SQL Property Graph Querying, TigerGraph, Memgraph, JanusGraph, DSE Graph, ...

Multiple languages

<table>
<thead>
<tr>
<th>ISO SC32.WG3</th>
<th>SQL PGQ (Property Graph Querying)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neo4j</td>
<td>openCypher</td>
</tr>
<tr>
<td>LDBC</td>
<td>G-CORE (augmented with paths)</td>
</tr>
<tr>
<td>Oracle</td>
<td>PGQL</td>
</tr>
<tr>
<td>W3C</td>
<td>SPARQL (RDF data model)</td>
</tr>
<tr>
<td>Tigergraph</td>
<td>GSQL</td>
</tr>
</tbody>
</table>

...also imperative and analytics-based languages

* http://www.opencypher.org/references#sql-pg

SQL 2020
Participation from major DBMS vendors.
Neo4j’s contributions freely available*.
A new stand-alone / native query language for graphs

Targets the labelled PG model

Composable graph query language with support for updating data

Based on

- “Ascii art” pattern matching
- Published formal semantics (Cypher, G-CORE)
- SQL PG extensions and SQL-compatible foundations (some data types, some functions, ...)

https://www.gqlstandards.org
GQL design principles

A property graph query language
GQL doesn’t try to be anything else

A **composable** language
Via graph projection, construction, subqueries
Closed under graphs and tables

A **declarative** language
Reading, updating and defining schema

An **intuitive** language

A **compatible** language: reuse SQL constructs where sensible, and be able to interoperate with SQL and other languages
GQL will be standardized under the aegis of ISO SC32/WG3

This is the body that specifies and standardizes SQL
SQL 2020 is currently being designed - includes SQL Property Graph Extensions

GQL will be specified as a separate language to SQL

Will incorporate features in SQL Property Graph Extensions as well as SQL functionality where appropriate

Goals:

Lead and consolidate the existing need for such a language
Increase utility of graph querying for ever more complex use cases
Covers full spectrum of features for an industry-grade graph query language
Drive adoption of graph databases
- Create, Read, Update, Delete (CRUD)
- Advanced complex path expressions
- Construct & project graphs
- Composable

- Academia Regular Queries

- W3C XPath
  - Extended by
  - Reading graphs

- Academia RPQs (Regular Path Queries)
  - Extended by
  - Reading graphs

- openCypher
  - Reading graphs
  - Complex path expressions
  - Create, Read, Update, Delete (CRUD)
  - Advanced complex path expressions
  - Construct & project graphs
  - Composable

- Oracle PGQL
  - Reading graphs
  - Complex path expressions
  - Read only
  - Path macro (complex path expressions)

- SQL PGQ
  - Reading graphs
  - Advanced complex path expressions
  - Create, Read, Update, Delete
  - Advanced complex path expressions with configurable matching semantics
  - Construct & project graphs
  - Composable

- GQL

- LDBC G-CORE
  - Reading graphs
  - Advanced complex path expressions
  - Creating, constructing and projecting graphs, Advanced complex path expressions, Composable

- Academia GXPath
  - Reading graphs
  - Complex path expressions
  - RPQs with data tests (node & edge properties)
Existing Languages Working Group (ELWG)

Interdisciplinary, independent group:
- Alin Deutsch (TigerGraph)
- Harsh Thakkar (University of Bonn (Germany))
- Jeffrey Lovitz (Redis Labs)
- Mingxi Wu (TigerGraph)
- Oskar van Rest (Oracle)
- Petra Selmer (Neo4j)
- Renzo Angles (Universidad de Talca (Chile))
- Roi Lipman (Redis Labs)
- Thomas Frisendal (Independent data modelling expert and author)
- Victor Lee (TigerGraph)

Goals:
- To construct a complete list/reference of detailed graph querying features
  - organised into feature groups
- To indicate, for each of these features, whether and how each language supports it
  - syntax and semantics

Languages:
- openCypher
- PGQL
- GSQL
- G-CORE
- SQL PGQ (Property Graph Querying)

Helping to facilitate the GQL design process

https://www.gqlstandards.org/existing-languages
Example GQL query

//from graph or view ‘friends’ in the catalog
FROM friends

//match persons ‘a’ and ‘b’ who travelled together
MATCH (a:Person)-[:TRAVELLED_TOGETHER]-(b:Person)
WHERE a.age = b.age
    AND a.country = $country
    AND b.country = $country

//from view parameterized by country
FROM census($country)

//find out if ‘a’ and ‘b’ at some point moved to or were born in a place ‘p’
MATCH SHORTEST (a)-[[:BORN_IN|MOVED_TO*]]-(p)<-[[:BORN_IN|MOVED_TO*]]-(b)

//that is located in a city ‘c’
MATCH (p)-[:LOCATED_IN]->(c:City)

//aggregate the number of such pairs per city and age group
RETURN a.age AS age, c.name AS city, count(*) AS num_pairs
GROUP BY age
GQL Features
Graph procedures

Inputs and outputs
- Graph
- Table
- Value

Nothing

GQL Function
- MATCH
- CONSTRUCT
- RETURN

Data Modifying Statement
- CREATE
- MERGE
- SET/REMOVE
- DELETE

Catalog-modifying GQL Procedure
- CREATE GRAPH
- DROP GRAPH

Procedures in any language

Graph Procedure

Modifying Graph Procedure

Modifying GQL Procedure

Reading Statement

Projecting Statement

GQL Procedures
Graph pattern matching
Patterns are everywhere

MATCH (query)-[:MODELED_AS]->(drawing),
  (code)-[:IMPLEMENTS]->(query),
  (drawing)-[:TRANSLATED_TO]->(ascii_art),
  (ascii_art)-[:IN_COMMENT_OF]->(code),
  (drawing)-[:DRAWN_ON]->(whiteboard)
WHERE query.id = $query_id
RETURN code.source

Expressed using “ASCII Art”

Patterns are in
● Matching
● Updates
● Schema (DDL)
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”
Complex paths in the property graph data model

Property graph data model:

*Properties* need to be considered

*Node labels* need to be considered

Specifying a cost for paths (ordering and comparing)

**Concatenation**
- \(a . b\) - a is followed by b

**Alternation**
- \(a | b\) - either a or b

**Transitive closure**
- \(\ast\) - 0 or more
- \(+\) - 1 or more
- \({m, n}\) - at least m, at most n

**Optionality:**
- \(?\) - 0 or 1

**Grouping/nesting**
- () - allows nesting/defines scope
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**

L. Libkin, W. Martens, and D. Vrgoč
Querying Graphs with Data
Composition of Path Patterns

Sequence / Concatenation:

\( \alpha \beta \)

Alternation / Disjunction:

\( \alpha | \beta \)

Transitive closure:

- 0 or more
- 1 or more
- n or more
- At least n, at most m

\( \alpha^* \)
\( \alpha^+ \)
\( \alpha^{n..} \)
\( \alpha^{n..m} \)

Overriding direction for sub-pattern:

- Left to right direction
- Right to left direction
- Any direction

\( \alpha > \)
\( \alpha < \)
\( \alpha > \)
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
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

\[
\text{PATH PATTERN } \text{road} = (a)-[\text{r:ROAD\_SEGMENT}]- (b) \quad \text{COST } \text{r.length}
\]

\[
\text{MATCH } \text{route} = (\text{start})-\text{\~road\*/-(end)}
\]

\[
\text{WHERE } \text{start.location} = \$\text{currentLocation}
\quad \text{AND} \quad \text{end.name} = \$\text{destination}
\]

\[
\text{RETURN } \text{route}
\]

\[
\text{ORDER BY cost(route) ASC LIMIT 3}
\]
Pattern matching today uses **edge 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)
MATCH (p:Person { name: 'Jack' })-[r1:FRIEND]-(friend)
MATCH (friend)-[r2:FRIEND]-(friend_of_a_friend)
RETURN friend_of_a_friend.name AS fofName

r1 and r2 may now be bound to the same relationship as they appear in two distinct patterns
Configurable pattern-matching semantics

Node isomorphism:
No node occurs in a path more than once
Most restrictive

Edge isomorphism
No edge occurs in a path more than once
Proven in practice

Homomorphism:
A path can contain the same nodes and edges more than once
Most efficient for some RPQs
Least restrictive
Path pattern modifiers

Controlling the path pattern-matching semantics

REACHES - return a single path, i.e. path existence checking

ALL - returns all paths

[ALL] SHORTEST - for shortest path patterns of equal length (computed by number of edges).

[ALL] CHEAPEST - for cheapest path patterns of equal cost, computed by aggregating a user-specified cost for each segment of the path

Other qualifiers

TOP <k> SHORTEST|CHEAPEST [WITH TIES] - only at most <k> of the shortest or cheapest possible paths

MAX <k> - match at most <k> possible paths

Some of these operations may be non-deterministic
Type system
Data types

Scalar data types
Numeric, string, boolean, temporal etc

Collection data types
Maps with arbitrary keys as well as maps with a fixed set of typed fields (anonymous structs):
{\text{name}: "GQL", \text{type}: "language", \text{age}: 0 }
Ordered and unordered sequences with and without duplicates: \[1, 2, 3\]

Graph-related data types
Nodes and edges (with intrinsic identity)
Paths
Graphs (more on this in the Schema section)

Sharing some data types with SQL’s type system

Support for
- Comparison and equality
- Sorting and equivalence
Advanced types

Heterogeneous types

MATCH (n) RETURN n.status  

Possible type system extension: Union types for expressing that a value may be one from a set of data types, e.g. A | B | NULL

Complex object types

Support the typing of complex objects like graphs and documents

Possible type system extension: Graph types, structural types, recursive document type

Static and/or dynamic typing

DYNAMIC  
Allow queries that may possibly fail at runtime with a type error

STRICT  
Reject queries that may possibly fail at runtime with a type error

Implementations may have different preferences
Expressions
Graph element expressions and functions

Element access: \texttt{n.prop}, \texttt{labels(n)}, \texttt{properties(n)}, ...

Element operators: \texttt{allDifferent(<elts>)}, =, <>

Element functions: \texttt{source(e)}, \texttt{target(e)}, (in|out)\texttt{degree(v)}

Path functions: \texttt{nodes(p)}, \texttt{edges(p)}, ...

Collection and dictionary expressions

- Collection literals: \texttt{[a, b, c, ...]}
- Dictionary literals: \texttt{\{alpha: some(a), beta: b+c, ... \}}
- Indexing and lookup: \texttt{coll[1]}, \texttt{dict['alpha']}
- More complex operations: map projections, list comprehension, etc
Schema and catalog
“Classic” property graphs: historically schema-free/optional

Moving towards a more comprehensive graph schema
- Label set - property mapping
- Extended with unique key and cardinality constraints
- Heterogeneous data

Partial schemas:
Data that doesn’t conform to the schema can still exist in the graph

Static, pre-declared portions alongside dynamically-evolving portions

Similar to Entity-Relationship (E-R) diagrams

I.e. the graph would be “open” with respect to the schema
Catalog

Access and manage multiple persistent schema objects

- Graphs
- Graph types (labels and associated properties)
- User-defined constructs: named graph procedures and functions
- Users and roles
Modifying and projecting graphs
Multi-part queries: reading and writing data

Modifying data operations
- Creating data
- Updating data
- Deleting data

Reading and writing statements may be composed linearly in a single query

```
FROM customers
MATCH (a:Person)
WHERE NOT EXISTS { (a)-[:HAS]->(:Contract) } 
WITH a, a.email AS email //query horizon
DETACH DELETE a
WITH DISTINCT email //query horizon
CALL {
  FROM marketing
  MATCH (c:Contact) WHERE c.email = email
  UPDATE marketing
  DETACH DELETE c 
}
RETURN email
```

- Follows established reading order in many languages
- Useful to return data reflecting the updates

Illustrative syntax only!
Graph projection

Sharing elements in the projected graph
Deriving new elements in the projected graph
Shared edges always point to the same (shared) endpoints in the projected graph
Projection is the inverse of pattern matching

**GRAPH MATCHING**

Original graph:

- (#1)→(#2)
- (#1)→(#3)
- (#3)→(#2)
- (#4)→(#2)

**SUBGRAPH MATCHES**

**DRIVING TABLE**

- a: #1, b: #2
- a: #1, b: #3
- a: #3, b: #2
- a: #3, b: #4
- a: #4, b: #2

**NEW ENTITIES**

- (#1)→[#5]→(#2)
- (#1)→[#6]→(#3)
- (#3)→[#7]→(#2)
- (#3)→[#8]→(#4)
- (#4)→[#9]→(#2)

**NEW GRAPH**

Graph construction:

- #1
- #2
- #3
- #4
- #5
- #6
- #7
- #8
- #9

Turns graphs into matches for the pattern

Turns matches for the pattern back into graphs
Query composition and views
Queries are composable procedures

- Use the output of one query as input to another to enable abstraction and views
- Applies to queries with tabular output and graph output
- Support for nested subqueries
- Extract parts of a query to a view for re-use
- Replace parts of a query without affecting other parts
- Build complex workflows programmatically
Implications

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

Construct new graphs from existing graphs

<table>
<thead>
<tr>
<th>a1</th>
<th>a1</th>
<th>a2</th>
<th>a2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
RETURN ... GRAPHS ...
```
Query composition and views

Disjoint base data graphs
“Sharing” of nodes and edges in **views**

A (graph) view is a query that returns a graph

Graph operations: `INTERSECT`, `UNION`, `UNION ALL`, ...

Support for **parameterized views**
Graph elements are shared between graphs and views
Graph elements have reference semantics and are ‘owned’ by their base graph or views that introduce them

Support for **updateable views**
Updates percolate downwards to the base graphs
Other work
Language mechanics

Interoperability between GQL and SQL
   Defining which objects in one language are available to the other

Interoperability with languages other than SQL

Security
   Reading and writing graph elements
   Executing queries

Error handling
   Failures and error codes
Future work

Graph compute and analytics
Session model and transaction semantics
Cursors
Constraints and triggers
Bidirectional edges
Stream processing
Multidimensional data
Temporal processing
To conclude...
GRAPH TECHNOLOGY LANDSCAPE 2019

https://graphaware.com/graphaware/2019/02/01/graph-technology-landscape.html

Image courtesy of Graphaware (esp. Janos Szendi-Varga)
Neo4j: Resources


Graph Databases (book available online at www.graphdatabases.com)

Getting started: http://neo4j.com/developer/get-started/

Online training: http://neo4j.com/graphacademy/

Meetups (last Wed of the month) at http://www.meetup.com/graphdb-london (free talks and training sessions)
Interested in joining the GQL design process?

Regular GQL Community Update Meetings

Working Groups

https://www.gqlstandards.org/

GQL Documents also available at http://www.opencypher.org/references#sql-pg
Thank you!