GQL (Graph Query Language)
Scope and Features

Graph Day, Austin, TX
26 January 2019

Petra Selmer
Query Languages Standards & Research Team at Neo4j

Dr. Petra Selmer*  
UK

Stefan Plantikow  
Germany

Tobias Lindaaker  
Sweden

Alastair Green  
UK

Dr. Hannes Voigt  
Germany

Also Dr. Peter Furniss  
UK
Agenda

Preliminaries:
- The property graph data model and graph querying basics using Cypher

Introducing Graph Query Language (GQL)

GQL Features
- Graph pattern matching
- Type system
- Expressions
- Tables
- Schema and catalog
- Modifying and projecting graphs
- Query composition and views
- Other work

Future work
Preliminaries
**Property graph**

Underlying construct is a **multigraph**, which may or may not be constrained by a schema.

Graph elements: **nodes** and **edges**

- **Nodes** (synonymous with *vertices*) with zero, one or more labels
- Typed, directed or undirected **edges** (synonymous with *relationships*) connecting any two nodes
- Zero, one or more **properties** (map containing key-value pairs) on nodes and edges

Guaranteed *topological consistency* (no dangling edges)

Every graph element:

- Has an intrinsic identity (e.g. two nodes could look identical in every other respect)
- Is owned by a single base graph
Property graph example

Person

name: Ed Jones
age: 37
favColor: blue

FRIENDS

since: 2003-05-12
qualified: good

Person

name: Peter Fry
nick: Lil' Pete
bio: ...

Director
Searching for (matching) graph patterns

MATCH (:Person { name:"Dan"}) -[:LOVES]-> ( whom ) RETURN whom
Graph querying basics using Cypher

Pattern Matching:
MATCH (me:Person)-[:FRIEND]->(friend)

Filtering:
WHERE me.name = "Frank Black"

Projection:
RETURN friend.name, friend.title

Data creation and manipulation:
CREATE (you:Person)
SET you.name = "Aaron Fletcher"
CREATE (you)-[:FRIEND]->(me)

Linear query composition:
MATCH (me:Person {name: $name})-[:FRIEND]->(friend)
WITH me, count(friend) AS friends
MATCH (me)-[:ENEMY]->(enemy)
RETURN friends, count(enemy) AS enemies

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

- Query processing begins at the top and progresses linearly to the end
- Each clause is a function taking in a table T and returning a table T'
- T' then acts as a driving table to the next clause
Graph querying basics using Cypher

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

Edge/relationship patterns:
MATCH ()-->( ), ()-[edge]->( ),
()-[edge:RELATES]->( ), ()-[:RELATES]->( ),
()-[edge {score:5}]->( ),
(a)-[edge]->(b), (a)<-[edge]-(b), (a)-[edge]-(b)

Variable-length paths:
MATCH (me)-[:FRIEND*1..3]-(foaf)

Returning paths:
MATCH p = (a)-[:ONE]-()-[:TWO]-()-[:THREE]-()
MATCH p = (me)-[:FRIEND*]-(foaf)
Introducing Graph Query Language (GQL)
Property graphs are everywhere

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

ISO SC32.WG3 \(\rightarrow\) SQL PGQ (Property Graph Querying)
Neo4j \(\rightarrow\) openCypher
LDBC \(\rightarrow\) G-CORE (augmented with paths)
Oracle \(\rightarrow\) PGQL
W3C \(\rightarrow\) SPARQL (RDF data model)
Tigergraph \(\rightarrow\) GSQL

...also imperative and analytics-based languages

SQL 2020
Participation from major DBMS vendors.
Neo4j’s contributions freely available*.

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

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

This is the first time this has happened in the history of the standardization of database languages
openCypher

- Construct & project graphs
- Composable

Cypher

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

Neo4j

Academia

- RPQs with data tests (node & edge properties)
- Read only
- Path macro (complex path expressions)

GXPath

Oracle

PGQL

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

SQL PGQ

GQL

LDBC

G-CORE

Academia Regular Queries
Existing Languages Working Group (ELWG)

Interdisciplinary, independent group:
- Alin Deutsch (TigerGraph)
- 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
Inputs and outputs
- Graph
- Table
- Value
- Nothing

Graph procedures
Graph pattern matching
Patterns are everywhere

Expressed using “ASCII Art”

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

Patterns are in
- Matching
- Updates
- Schema (DDL)
Finding and returning complex paths with path patterns

Based on (conjunctive) regular path queries (CRPQs)

\[ X, \left( \text{likes.hates} \right) * \left( \text{eats|drinks} \right) +, Y \]

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

Property graph data model:

Properties need to be considered

Node labels need to be considered

Specifying a cost for paths (ordering and comparing)

Plenty of academic research in this area since 1987!
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

Allow all three types of matching
All forms may be valid in different scenarios
Can be configured at a query level
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):
  `{name: "GQL", type: "language", 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 may give conflicting types (esp. in a large schema)

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: \(n.\text{prop}, \text{labels}(n), \text{properties}(n), \ldots\)

Element operators: \(\text{allDifferent(} \langle \text{elts} \rangle \rangle, =, \neq\)

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

Path functions: \(\text{nodes}(p), \text{edges}(p), \ldots\)

Collection and dictionary expressions
  - Collection literals: \([a, b, c, \ldots]\)
  - Dictionary literals: \(\{\text{alpha: some}(a), \text{beta: } b+c, \ldots\}\)
  - Indexing and lookup: \(\text{coll}[1], \text{dict}[\text{‘alpha’}]\)
  - More complex operations: map projections, list comprehension, etc
Tables
Why tabular projections in GQL?

Pattern matching => (Multi) set of bindings (=> Table)

Bindings main input into graph modifying operations (DML)
  Supported by tabular result transformation and combination

Bindings main input into graph construction operators
  Supported by tabular result transformation and combination

Features focused on tables as a foundational data model - e.g. referential integrity via FK - will not be considered

A “driving table”
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**: Turns graphs into matches for the pattern.
- **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**: 
  - #1
  - #2
  - #3
  - #4
  - #6
  - #7
  - #8
  - #5
  - #9

- **Original Graph**:

  - #1
  - #2
  - #3
  - #4

- **Subgraph Matches**:
  - (#1)→(#2)
  - (#1)→(#3)
  - (#3)→(#2)
  - (#3)→(#4)
  - (#4)→(#2)
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
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
GQL - Graph Query Language

A new and independent

Declarative,

Composable,

Compatible,

Modern,

Intuitive

Property Graph Query Language
Resources

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

petra.selmer@neo4j.com