GQL Scope and Features

Digest of BNE-023
with references to relevant sections

Neo4j Query Languages Team, WG3 Meeting, Brisbane, Australia, 2019
Paper outline

① Introduction
Inputs: Cypher, PGQL, SQL, G-Core, Motivation, Orientation, Concordances to [ERF-031] and [BNE-030] - [1.4]

② References
Related material, including design documents from the openCypher process

③ Discussion
Overarching design principles, Language overview

④ Proposal
Project scope, definitions, more detailed language features

⑤ Grammar
Sketch of proposed syntax - to show structure
Motivation [3.9, 3.10]

1. Lead and consolidate the existing demand for such a language
2. Address the specific needs of graph use cases
3. Increase the utility of property graph querying
4. Drive adoption of graph database systems
Motivation [3.9, 3.10]

1. Lead and consolidate the existing demand for such a language
2. Address the specific needs of graph use cases
3. Increase the utility of property graph querying
4. Drive adoption of graph database systems

A new and independent

- Declarative,
- Composable,
- Compatible,
- Modern,
- Intuitive

Property Graph Query Language
Inputs

- **BNE-023** uses Cypher, PGQL, SQL/PG, G-Core as main inputs
- Considers other languages as well
- Picked keywords and operators reflecting common use (some overlap already with e.g. **BNE-035**)
- Follows top-down approach structurally as a base line
- This needs more exploration and discussion
Example query [3.1]

// from graph or view "friends" in the catalog
FROM friends

// match persons a and b that travelled together
MATCH (a IS Person)-[IS TRAVELLED_TOGETHER]-(b IS 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 where born in a place p
MATCH SHORTEST (a) (()-[IS BORN_IN|MOVED_TO]->())* (p)
  (())<-[IS BORN_IN|MOVED_TO]-(())* (b)

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

// aggregate number of such pairs per city and age cohort
RETURN a.age AS age, c.name AS city, count(*) AS pairs GROUP BY age
GQL Design Principles [3.9]

① **New and independent property graph query language**

*Follow the tradition of existing languages like Cypher, PGQL and SQL/PGQ*

*Support both standalone implementation or extension of existing SQL-based systems*

② **Declarative language**

*Emphasize what over how (in particular by using pattern matching)*

③ **Composable language**

*Compose procedures from nested sub-procedures and statement sequences*

④ **Compatible language**

*Ensure compatibility with established and widely used features of SQL and avoid idle variation to existing syntax*

⑤ **Modern language**

*Introduce next-generation language features using established designs from existing languages where available*

⑥ **Intuitive language**

*Follow consistent, "whiteboard-friendly", visual syntax (in particular by using "ascii-art" patterns)*
Scope Overview [4.1]

The GQL language is a composable declarative database language for querying and managing property graphs that is intended to be useable both as an independent language as well as in conjunction with SQL or other languages.

① Composable and closed under graphs and tables

② Declarative (i.e. independently implementable using different strategies)

③ Covering the full spectrum of features of an industry-grade database query language (next slide)
Scope Overview

1. Graphs & Pattern matching
2. Tables & Expressions
3. Type system & Schema
4. Modifying and Projecting graphs
5. Query composition & Views
6. Schema & Catalog
7. Interoperability
8. Error handling model
9. Security model
10. User defined procedures & functions
Graphs & Pattern matching [3.4, 4.6]

1. Property graph model
2. Nodes, Edges, and Paths [4.4.3]
3. Patterns [ERF-035, BNE-034]
5. Working with paths [4.6.5]
Property Graph Model

- **Vertex|Node**
  - 0-* Labels
  - key:value properties
  - Intrinsic identity

- **Edge|Relationship**
  - 0-* Labels
  - key:value properties
  - Related vertices/nodes
    - requiring referential integrity
  - Directed or Undirected
  - Intrinsic identity
    - (might depend on the nodes)

- Every graph element is owned by a single (base) graph

- Intrinsic identity allows multiple vertices or relationships with the same labels or properties but potentially different topology

- Topological consistency necessary, i.e. no dangling relationships allowed
  => Isolation requirement *(not discussed here)*

- Graphs may be constrained by schema *(not discussed here)*

- Paths are sequences of alternating vertices (1 .. n+1) and edges (0 .. n) that are connected in the graph, without gaps, starting and ending in nodes
Intrinsic Identity
Graph Patterns

MATCH (query)-[MODELED_AS]->(drawing),
    (code)-[IMPLEMENTED]->(query),
    (drawing)-[TRANSLATED_TO]->(ascii_art),
    (ascii_art)-[IN_COMMENT_OF]->(code),
    (drawing)-[DRAWN_ON]->(whiteboard)
WHERE query.id = {query_id}
RETURN code.source
Pattern matching structure

[FROM <graph>]
MATCH <pattern> {<comma> <pattern> ...}

+ optional modifiers to MATCH
for controlling pattern matching behaviour

OPTIONAL MATCH  - outer join, binds nulls if nothing matches
MANDATORY MATCH - query fails if nothing matches

FROM twitter
MATCH (a)-[IS Follows]->(b)

OPTIONAL MATCH (  
  (b)-[p IS Posted]->(m)  
  WHERE p.date > three_days_ago  
)

MATCH ...
- DIFFERENT (VERTICES|NODES) - vertex isomorphism
- DIFFERENT (EDGES|RELATIONSHIPS) - edge isomorphism
- UNCONSTRAINED - homomorphism
Pattern matching modifiers

<path modifiers> for controlling path matching semantics

[ALL] SHORTEST - for shortest path patterns
[ALL] CHEAPEST - for cheapest path patterns (both with TOP <k>, MAX <k> qualifiers, and supporting WITH TIES)

REACHES - unique end nodes with >=1 matching path
ALL - all paths

SIMPLE - may not contain repeated nodes
TRAIL - may not contain repeated edges
ACYCLIC - may not repeat nodes, except allowing the first and last node to be the same

FROM twitter
MATCH SIMPLE (a) (()-[IS Knows]->())* (b),
    TRAIL (a)-[IS Lives_At]->()
    (()-[IS Bus|Train|Plane]->())*
    ()<-[IS Lives_At]-(b)
Tables & Expressions [4.5, 4.9]

1. Basic table operations (selection, projection, ordering, filtering, slicing) [4.9]
2. Aggregation and grouping [4.5]
3. Tabular set operations (UNION [ALL]) [4.9]
4. Graph element expressions [4.5]
5. Collection and dictionary expressions [4.5]
6. Relationship to SQL
Why tabular operations in GQL?

(A) Pattern matching => (Multi) set of bindings (=> Table) => Tabular result transformation useful to avoid client-side processing

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

(C) Bindings main input into graph construction operators => Supported by tabular result transformation and combination

Not needed: Features focussed on tables as a base data model like e.g. referential integrity via foreign key constraints
Tabular projection [4.5, 4.9]

- Projecting values from the graph elements bound by pattern matching (incl. `DISTINCT`)
- Tabular aggregation (sum, count, avg, ...) and grouping (incl. on graph elements and perhaps window functions)
- Sorting, slicing

... Like `SELECT` in SQL

```sql
FROM social_graph
MATCH (me IS PERSON)
  -[IS FRIENDS]-(
  -[IS FRIENDS]-(foaf)
WHERE me.age > 18
  AND NOT EXISTS (me)-[IS FRIENDS]-(foaf)
RETURN
  me.name AS Name,
  array_agg(foaf.name) AS Suggestions
GROUP BY me
```

Additionally:
- Table combinations (like `UNION`, etc.)
- Collection unnesting (`UNNEST`)
- Simple lateral joins
Graph element expressions and functions

- Element access: `n.prop`, `labels(n)`, `properties(n)`, `handle(n)`
- Dynamic label tests
- Element operators: `allDifferent(<elts>), =, <>`
- Element functions: `source(e)`, `target(e)`, `(in|out)degree(v)`
- Path functions: `nodes(p)`, `edges(p)`, `concatenation`
Collection and dictionary expressions

- Collection literals: \[ a, b, c, \ldots \]  
- Dictionary literals: \{ alpha: some(a), beta: b+c, \ldots \}  
- Indexing and lookup: coll[1], dict[‘alpha’]  
- Map comprehensions  
- List comprehension  
- Functions
Type system & Schema [3.3, 4.4]

1. Selected scalar data types from SQL [4.4.1]
2. Nested data and collections [4.4.2]
3. Graph-related data types [4.4.3]
   - Nodes and Edges - with intrinsic identity
   - Paths
   - Graphs
4. Advanced type system features [3.3, 4.4.4]
5. Static and dynamic typing [4.4.5]
Advanced types

Heterogeneous types
MATCH (n) RETURN n.status may give conflicting types (esp. in a large schema)
Possible type system extension: Union types, e.g. A | B | NULL

Partial/incomplete types
Data access (e.g. in a big data file system) with runtime metadata discovery
Possible type system extension: Gradual type for "value of unknown type"?
Possible type system extension: Open graph element types..

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

- It is being proposed that GQL will have a rich schema
- Lends itself naturally to type and label inference, type checking, data flow analysis (e.g. reasoning about labels across **CASE** when using label tests with flow typing)
- Providing advanced type system features statically may prove to be challenging
- Different implementations may have different preferences

Possible solution: Support both modes with **DYNAMIC** being mandatory
Modifying and Projecting graphs

① Modifying graphs using patterns [3.5, 4.7]
② Graph projection [4.8]
③ Element sharing [4.8]
④ Graph combinators (UNION, INTERSECTION, ...) [4.8]
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
Graph Projection is the inverse of pattern matching
Graph Projection is the inverse of pattern matching

**ORIGINAL GRAPH**

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

**SUBGRAPH MATCHES**

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

**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]-(#1)
- (#1)<-[#7]-(#2)
- (#1)<-[#8]-(#4)
- (#4)<-[#9]-(#2)

**NEW GRAPH**

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

**GRAPH CONSTRUCTION WITH GROUPING**
Query composition & Views

1. Composable graph procedures [4.3.3-4.3.5]
2. Parameters and results [4.3.2, 4.3.4.1]
3. Linear statement composition [3.10.3, 4.3.4.3]
4. Graph views model [3.7]
5. Updatable Views and Graph augmentation
6. Provenance tracking
Queries are procedures [4.3]

- Use the output of one query as input to another to enable abstraction and views.
- Both for queries with *tabular* output and *graph* output.
- Nested queries and procedures [4.10].
- Simple linear composition of tabular output of one query as input to another [3.10.3].
Graph Procedures

Inputs/Outputs:

- Graph
- Table
- Value
- Nothing
Linear statement composition [3.10.3, 4.3.4.3]

- Top-Down flow
- Combined using lateral join
- Statements are update horizons

Benefits
- Natural, linear order used in programming
- Allows query-aggregate-query without (named) nested subqueries
- Allows mixing reading and writing (e.g. returning modified data)
- Solvable using subquery unnesting (maps on "apply" operator)
- Very positively received by users
Linear statement composition [3.10.3, 4.3.4.3]

- Top-Down flow
- Combined using lateral join
- Statements are update horizons

Syntax

- `SELECT` vs `RETURN`?
- `SELECT` and `RETURN`?
- This needs further discussion and investigation
Combining composition, projection, and linear flow

QUERY sameCityFriends {
  MATCH (a)-[e1:LIVED_IN]->(c:City)<-[e2:LIVES_ID]-(b)
  WHERE EXISTS (a)-[:KNOWS]-(b) AND e1.year = e2.year
  CONSTRUCT
  MERGE (a), (b)
  CREATE (a)-[:SAME_CITY_FRIEND]-(b)
  RETURN GRAPH
}
FROM sameCityFriends
MATCH (a)-[:SAME_CITY_FRIEND]-(x)-[:SAME_CITY_FRIEND]-(b)
WHERE a <> x AND x <> b AND a <> b
RETURN a.name, count(b) AS num_same_city_friend_of_a_friend
Graph elements are shared between graphs and views
- Graph elements are "owned" by their base graph or introducing views
- Sharing graph must form a DAG
Views [3.7, 4.12]

- A (graph) view is a query† that returns a graph or a view is the graph returned by a query†
  - GQL could also support tabular views - a query† that returns a table

- A view can be used as if it was a graph (a tabular view can be used as if it was a table)

- Queries† can be parameterized, so can views
  - allowing the application of the same transformation over any (compatible) graph

† procedure
Updatable Views

Updatable views = Track graph element origin across projecting views

- for shared elements trivial due to intrinsic identity
- for properties similar to tabular views in SQL

Updatable views

- Updates simply trickle down
- Similar problems regarding creation and deletion
  - What is the target base graph for a newly created element?
  - How to deal with dangling relationships when deleting? Cascade?
Views behave as if conceptually computed on the fly, including shared graph elements.

What if one wants to explicitly express persistently shared graph elements?
Graph Augmentation

Graph augmentation: Allow explicit persistent layered graphs with shared graph elements

Many open questions:
- Deletion semantics
- Combining graph projection and augmentation
Provenance Tracking

Graph elements implicitly have **provenance**

- shared elements with same identity: owning graph
- projected elements (with different identity): contributing elements

**Provenance tracking:**
Explicitly expose provenance information

**Use:**
Re-consolidate elements from multiple views or augmenting graph with same provenance
Language mechanics

1. Schema & Catalog [3.8, 4.11]
2. Language Interoperability [4.13]
4. Error handling model [4.15]
GQL Graph Query Language

A new and independent

Declarative,

Composable,

Compatible,

Modern,

Intuitive

Property Graph Query Language