Graph Query Language Task Force
first year update

Peter Boncz
boncz@cwi.nl
GraphQL Background

The GraphQL task force of LDBC studies query languages for graph data management systems, and specifically those systems storing so-called Property Graph data.

This query language should cover the needs of the most important use-cases for such systems, including (at least) the LDBC's own social network benchmark's Interactive and Business Intelligence workloads.
GraphQL TF Composition

- Renzo Angles, Universidad de Talca
- **Marcelo Arenas, PUC Chile - task force lead**
- Pablo Barceló, Universidad de Chile
- Peter Boncz, Vrije Universiteit Amsterdam
- George Fletcher, Eindhoven University of Technology
- Irini Fundulaki, FORTH
- Claudio Gutierrez, Universidad de Chile
- Tobias Lindaaker, Neo Technology
- Marcus Paradies, SAP
- Raquel Pau, UPC
- Arnau Prat, UPC / Sparsity
- Tomer Sagi, HP Labs
- Oskar van Rest, Oracle Labs
- Hannes Voigt, TU Dresden
- Yinglong Xia, Huawei America
the goals of the GraphQL task force are the following:

• to devise a list of **desired features** and **functionalities** of such a query language

• to evaluate a number of existing languages, in particular Cypher, and possibly Gremlin v3, SPARQL and SQL in this respect and identify possible problems in these.

• The result should be a better understanding of the design space and state-of-the-art.

• The target is to achieve this **within one year**. In a second phase, we can develop proposals for changes to existing query languages, or even a new query language..
GraphQL Log (14/28)

2015-06-08 wiki, data model
2015-06-22 data model
2015-07-06 data model
2015-07-22 case study: SPARQL 1.1
2015-08-03 case study: Cypher
2015-08-17 case study: PGQL
2015-08-31 theory: Regular Path Queries
2015-09-28 case study: Sparksee API
2015-10-12 case study: Gremlin
2015-10-26 survey on history of graph query languages
2015-11-16 survey on history of graph query languages
2015-11-23 case study: “graphs at a time” proposal sigmod2008
2015-12-07 case studies: conceptual schemas (i), and composability (ii)
2015-12-21 summary so far, attention for LDBC SNB query requirements
2016-01-11 (i) LDBC TUC use case overview, (ii) types (graphs, tables, paths)
2016-01-25 case studies: type systems in Cypher and PGQL
2016-02-01 meta-discussion: wiki pages for graph data model, functionalities
2016-02-15/02-29/03-07 generate more examples and functionalities
2016-03-14 case study: graph pattern matching & binding tables
2016-03-22 discussion: binding tables ➔ without schema
2016-04-04 proposal: reachability queries
2016-04-18 discussion: shortest path queries ➔ monotone top-k with constraints
2016-05-09 proposal: RPQs with regular expression with memory (REM)
2016-05-23 proposal: relational graph query processing (aka Peter’s brain dump)
2016-05-30 proposal: constraints on paths
2016-06-06 discussion: Peter’s brain dump conclusions
2016-06-20 proposal: data type transformations
In the following definition, we assume the existence of the following sets:

- \( \mathbf{L} \) is an infinite set of (node and edge) labels;
- \( \mathbf{P} \) is an infinite set of property names;
- \( \mathbf{V} \) is an infinite set of literals (actual values).

Moreover, we assume that \( \text{SET}(X) \) is the set of all finite subsets of a given set \( X \). Then a property graph is a tuple \( G = (N, E, \rho, \lambda, \sigma) \), where:

- **nodes**: \( N \) is a finite set of nodes;
- **edges**: \( E \) is a finite set of edges such that \( N \) and \( E \) have no elements in common;
- \( \rho : E \rightarrow (N \times N) \) is a total function;
- **labels**: \( \lambda : (N \cup E) \rightarrow \text{SET}(\mathbf{L}) \) is a total function;
- **properties**: \( \sigma : (N \cup E) \times \mathbf{P} \rightarrow \mathbf{V} \) is a partial function.

We decided not to define a schema (expected properties and their types, given a label).
What are the types needed in the graph query language, apart from the basic types (such as string and integer)?

- It has been argued that GRAPH and TABLE should be types in the languages.
- It has also been argued that a type PATH should be included in the language.
- Do we need to consider only simple paths?
- Do we need to consider sets of objects? E.g. return a set of graphs.
- Do we need to include lists of objects? E.g. a path could be a list of vertexes.
Discussion: Shortest Paths Functionality

- shortest paths (hops), and/or weighted shortest path
  - weight function: monotone sum (only then Dijkstra)
- path constraints (and implications for efficiency)
  - Constraints on what? Just {edge, vertex} properties on the path?
  - Or full-blown subqueries? Constraints involving the path so far?
- query embedding of shortest paths
  - single shortest paths (between one source and destination)
  - Or: all pair shortest paths
  - Or: bulk shortest paths (between many src, dst combinations, e.g., delivered by subquery)
- What to return:
  - The distance / total weight?
  - Or the shortest path? What if multiple path with the same cost exist? Return one or multiple, and if so, how to make this deterministic?
- top-N shortest paths – a natural extension of shortest paths (N=1)
  - Best N paths for each src, dst pair.
  - Is this useful functionality? Some use cases cast doubt on this
Relational Graph Querying

Idea: “seeing graphs in tables”

- $G = (V,E)$ with
  - $V$ denoting a table of vertexes, with
    - one non-null unique key column $V.key$
    - nullable columns $V.p_i$ holding vertex properties $p_i$;
  - $E$ denoting a table of edges with
    - columns $E.from$ and $E.to$ holding non-null values from the domain of $V.key$
    - nullable columns $E.p_j$ holding edge properties $p_j$
- We can use non-NF1 tables for multi-valued properties
- There are two foreign key constraints
  - $E.from \rightarrow V.key$
  - $E.to \rightarrow V.key$

```
ALTER TABLE E ADD GRAPH KEYS (mykey)
EDGE (from) TO (to)
REFERENCES V(key)
```
Example SQL Extension

On to cheapest weight path queries:

```
SELECT v1, v2, CHEAPEST SUM(e:distance) score, ..
FROM .. (introducing v1 and v2 here) ..
WHERE v1.key REACHES v2.key OVER E e EDGE E_from,E_to
ORDER BY ..
```

- Rule: if a CHEAPEST SUM(X:) predicate is used in the SELECT list, this must match a REACHES..OVER X condition in the WHERE, in which case we do not only ask to filter where paths exists, but also compute the cheapest cost of all such paths (this cost is bound to score).
- The parameter to SUM(X:expr) can be a complex expr, in which (only) binding variable X can play a role. Note that it may be used to access edge properties.
- Note we avoid binding a variable to the space of all possible paths in this syntax.
- Restricting bindings of e to only the edges on the single-cheapest path (for each v1,v2) is healthy as I have become convinced that top-N paths only produce meaningless results on real data, with N>1
Decisions: Relational Graph Querying

(1) Using tables to represent vertexes, edges, and paths
   • Accepted.

(2) Using nested tables to represent paths
   • Accepted.

(3) Constructing edge sets from subqueries, i.e., having compositionality of queries
   • Accepted

(4) Restricting to monotone sums for weighted shortest path functions (accepted)
   • Accepted

(5) Using a black box approach to shortest paths that avoids exposing all path bindings
   • No conclusion yet

(6) It is a worthwhile/positive endeavor to consider extending SQL, in addition to design of native graph QL.
   • Accepted to do a coupled joint study of two languages.
Computable Path Constraints: REM

- $k$ registers $x_1,...,x_k$ that store property/value pairs
- Conditions: Boolean condition $c$ that compares property/value pairs in node currently visited with the ones stored in the registers (e.g., $x_3 = \text{current.prop1 AND } x_{11} > \text{current.prop3}$)
- Extend usual regexps with:
  
  \[
  e[c] \quad \text{and} \quad p \rightarrow x \$ e
  \]
  
  - $e[c]$: read path according to $e$ and check that condition $c$ holds over its last node
  - $p \rightarrow x \$ e$: store value of property $p$ of the first node in register $x$ and check that the rest of the path satisfies $e$

Proposal gets a lot of expressive power out of the efficiently computable family
Proposal is criticized for being hard to understand by non-expert users
Composable Graph Patterns

In addition to *graph* patterns, allow for specifying *path* patterns with vertices, edges and constraints:

- Path with edge label constraints:
  
  ```
  ```

- Path with vertex label constraints:
  
  ```
  PATH 1123 := (:L1) -> (:L2) -> (:L3)
  ```

- Path with property (and label) constraints:
  
  ```
  PATH ab := () -> [:a] -> () -> [:b] WITH \( p1 > 3 \) AND \( p2 < 4 \) -> ()
  ```

- Path with cross-constraints:
  
  ```
  PATH ab := (x) -> [:a] -> (y) -> [:b] -> (z)
  WHERE \( y.p1 > x.p1 \) AND \( z.p1 > y.p1 \)
  ```
Composable Graph Patterns

Path pattern composition

• A graph pattern in the WHERE clause can be composed of path patterns:

```plaintext
PATH abc := () -[:a]-> () -[:b]-> () -[:]-> ()
SELECT y
WHERE (x@123) -/:abc*/-> (y)
```

  Find all vertices $y$ reachable from vertex $x$ with identifier 123, via an $abc^*$ path

• Specify repeated application of path patterns using the Kleene star

• A path pattern can be composed of other path patterns (to support nested Kleene star)

```plaintext
PATH redEdge := () -[e WITH color = ‘RED’]-> ()
PATH manyRedOneBlue := () -/:redEdge*/-> () -[e WITH color = ‘BLUE’]-> ()
SELECT y
WHERE (x@123) -/:manyRedOneBlue*/-> (y)
```
Composable Graph Patterns

Returning paths

• Return a single min-hop shortest path for each source-destination pair \((k = 1)\)

```
PATH abc := () -[:a]-> () -[:b]-> () -[:c]-> ()
SELECT p
WHERE (x@123) -/p:abc*/-> (y)
```

• Return \(k\) min-hop shortest paths for each source-destination pair \((k = 30)\)

```
PATH abc := () -[:a]-> () -[:b]-> () -[:c]-> ()
SELECT p
WHERE (x@123) -/p:abc*/#30-> (y)
```
Decisions: Composable Path Patterns

The graph query language should allow for:

1. (node-selecting) reachability RPQs
   • Accepted.

2. k-shortest path finding RPQs (i.e., path-selecting queries)
   • Accepted.

3. Constraining both edge labels and properties of vertices and edges along paths.
   • Rejected.

4. Comparing data values (labels/properties) along paths
   • Accepted.

5. Translation of all PQs to REMs ("queries should be executable in polynomial time")
   • Accepted.

6. Specifying min+max repetition on Kleene stars
   • Accepted.
Discussion & Outlook

• Did we achieve our year#1 objectives?
  – We got close.
  – Some really great people in the TF. Good atmosphere.

• Modus Operandi of GraphQL TF
  – Not easy to structure such a multi-faceted discussion
  – Linear decision points?

• Future
  – More {discussions, case studies, functionalities, *}
  – A language proposal document
    • One proposal, or two (native + SQL extension)?