CWI
Centrum Wiskunde & Informatica

github.com/ldbc/g-core

G-CORE

The LDBC Graph Query Language Proposal

LDBC GraphQL Task Force:

Marcelo Arenas, Pablo Barcelo (Universidad Catolica)
Alastair Green, Stefan Plantikow, Tobias Lindaaker (neo4j)
Marcus Paradies (SAP)
Oskar van Rest (Oracle Corp.)
Arnau Prat(Sparksee)
George Fletcher (TU Eindhoven)
Hannes Voigt (TU Dresden)
Renzo Angles (U Talca)
Peter Boncz (CWI)

LDBC

What, How and Where

- Goal:
  - Recommend a query language core that could be incorporated in future (versions) of industrial graph query languages.
  - Perform deep academic analysis of the functional and complexity space to ensure a powerful and practical language.

- 2 years of work in the GraphQL task force
- 3-day workshop in Santiago de Chile (Aug 8-10)
  - Lots of progress, this proposal.
- Writing workshop for SIGMOD industrial paper (Dagstuhl Nov 26- Dec 1)

Principles & Goals

- Closed Query Language
  - to allow subqueries, views
  - Define PG property graphs as
    - “directed, labeled graphs with paths, and properties on all of these”

PG Data Model

- Property Graph data model has never been formally defined
- We define it here:
  - **L** is an infinite set of label names for nodes, edges and paths;
  - **K** is an infinite set of property names;
  - **V** is an infinite set of literals (actual values).

Moreover, given a set X, we assume that SET(X) is the set of all finite subsets of X (including the empty set), and LIST(X) is the set of all finite lists of elements from X.

Then a property graph is a tuple $G = (N, E, P, \rho, \delta, \lambda, \sigma)$, where:

- N is a finite set of nodes.
- E is a finite set of edges.
- **P is a finite set of paths. We assume that N, E and P are pairwise disjoint.**
- $\rho : E \rightarrow (N \times N)$ is a total function.
- **$\delta : P \rightarrow$ LIST($N \cup E$) is a total function. We assume that for every path $p \in P$, it holds that either $\delta(p) = [a]$ with $a \in N$ or $\delta(p) = [a_1, e_1, a_2, \ldots, a_n, e_n, a_{n+1}]$,**
  1. **$n \geq 1$,**
  2. **$a_j \in N$ for every $1 \leq j \leq n+1$,**
  3. **$e_j \in E$ for every $1 \leq j \leq n$, and (iv) $\rho(e_j) = (a_j, a_{j+1})$ or $\rho(e_j) = (a_{j+1}, a_j)$ for every $1 \leq j \leq n$**
- $\lambda : (N \cup E \cup P) \rightarrow$ SET(**L**) is a total function.
- $\sigma : (N \cup E \cup P) \times$ **K** $\rightarrow$ SET(**V**) is a total function. We assume that there is a finite set of tuples (a,b) in $(N \cup E \cup P) \times$ **K** such that $\sigma(a,b)$ is not the empty set.

Principles & Goals

- Closed Query Language
  - to allow subqueries, views
  - Define PG property graphs as
    - "directed, labeled graphs with paths, and properties on all of these"
- Powerful Language Features
  - PATH … → specifies path patterns, (weighted) shortest path finding types
    - Also ALLPATHS, using PROJECT GRAPH to return all matching edges as a graph
  - MATCH … → graph matching resulting in bindings
  - WHERE … → specifies filtering
  - CONSTRUCT GRAPH … → project bindings into a graph (incl grouping)
  - Full query: e.g. UNION of multiple CONSTRUCT GRAPH returning one graph

Notation

- x variable
- (x) node
- [y] edge
- /p/ path

Given a variable x, we use notation

- x:L L is a label of a node, edge or path x
- x:L { p = v } the value of property p is v for x

In regular expressions:

- (r)* Kleene star,
- (r1 + r2) union
- (r1 r2) concatenation

MATCH

binds variables using homomorphic pattern matching, it results in an (imaginary) binding table where the columns are the variables and the rows the bindings.

- **MATCH** (a:Person {age=21})-[:knows]->(b)-[:knows]->(c)
- **MATCH** (a:Person {age=21})-/<knows*>/->(b)-/<knows*>/->(c)
- **MATCH** (a:Person {age = 21})-/p<friend*>/-(b:Person {age = 21})
- **MATCH** (a)-/p<knows*> {confidence = 0.9}/-(b)

distinguish between an edge label and a regular expressions composed by a single edge label:

- **MATCH** (a)-/p<knows>/-(b) // p binds to a path object
- **MATCH** (a)-[p:knows]-(b) // p binds to an edge object

checking multiple labels by using notations (x:L1:L2), [y:L3:L4:L5] and /z:L6:L7/. The semantics of these expressions is conjunctive;

Variables are typed depending to which kind of object (node, edge, or path) they are assigned. Invalid: (x)-[x]-(y)

MATCH …OPTIONAL … ON

- **Optional match.** Inside the sequence conjuctive. Variables are NULL if not matchable.
  MATCH p_1, p_2, ..., p_k **OPTIONAL** q_1, q_2, ..., q_l **OPTIONAL** r_1, r_2, ..., r_m . . .
  where each p_i, $q_{i,}$ and r_i is a basic graph pattern. Example

  MATCH (x)-[:name]->(y) **OPTIONAL** (x)-[:phone]->(z)

  There is only one level of nesting for the **OPTIONAL** operator.

- **Graph source**. The graph the matching operates on can be specified by **ON** after a basic match pattern. The specification is optional. If no graph is specified the matching will operate on the base graph. If a graph is specified, the matching will operator on that graph. Graphs can be specified by an identifier for named graphs or by a subquery.

  MATCH (x)-[:name]->(y) **ON** Employees

Expressions

- Local connectives: **AND**, **OR**, **NOT**
- Grouping: parenthesis
- Standard Functions: **abs**(x), **sin**(x), ...
- Graph Functions: **labels**(n), **nodes**(p), **rels**(p), **startNode**(r), **endNode**(r), **length**(p) **size**(l), **bound**(n)
- Label-Test:
  - n:Person **true** iff n has the Person label
  - n:_ **true** iff n has any label
- Collections:
  - Membership test elem **IN** collection
  - Indexing collection[elem]
  - Slicing collection[slice]
  - List comprehension [x **IN** list **WHERE** predicate | expr]

WHERE

Filters bindings resulting from MATCH.

Operators with boolean result:

- =, <>,
- <, <=, >, >=
- +, -, *, /, %, ...
- v.prop (where v is a node/edge/path variable)
- CASE

Subqueries:

- EXISTS { <full-query> }
- Can refer to variables from outer scope

Shorthand for dealing with expressions on set-valued properties?

- Startswith((person.phone), “+1”)
- If c.phone is multi-valued: ERROR (or EXISTS/ALL semantics?)

Existential WHERE match

Examples.

MATCH (x)-[:friend]->(y)-[:friend]->(z) **WHERE** (x)-[:friend]->(z)
is equivalent to:
MATCH (x)-[:friend]->(y)-[:friend]->(z) **WHERE EXISTS { MATCH** (x)-[:friend]->(z) }

Negation:

MATCH (x)-[:friend]->(y)-[:friend]->(z) **WHERE NOT** (x)-[:friend]->(z) is equivalent to:
is equivalent to:
MATCH (x)-[:friend]->(y)-[:friend]->(z) **WHERE NOT EXISTS { MATCH** (x)-[:friend]->(z) }

WHERE clause does not introduce new bindings in the outer scope.
It just filters **MATCH** bindings.

PATH

- syntactically starts on the left with a vertex,ends at its rightmost vertex
  - **PATH** abc **AS** ()-[a:A]-()-[b:B]-()
- the pattern can be augmented to be nonlinear, by comma
  - **PATH** abc **AS** (src)-[a:A]-()-[b:B]-(dst), (dst)-[c:C]-()
- can have a **COST** clause:
  - **PATH** abc **AS** ()-[a:A]-()-[b:B]-()-[c:C]-()
    COST a.x + b.y + c.z
- can have a WHERE clause:
  - **PATH** unreciprocated_love **AS** (a) -[:LOVES]->(b)
    WHERE NOT (b)-[:LOVES]->(a)

  which is equivalent to:
  - **PATH** unreciprocated_love **AS** (a) -[:LOVES]->(b)
    WHERE NOT EXISTS { **MATCH** (b)-[:LOVES]->(a) }

PATH usage in MATCH

Existence of paths (no path variable binding)

- **MATCH** (x)-/~friend*/-(y)

Shortest path queries - a fixed number of shortest paths

- **MATCH** ()-/**SHORTEST** p:~friend*/-() (single shortest path)
- **MATCH** ()-/5 **SHORTEST** p:~friend*/-() (5 shortest path2)

Weighted shortest path:

- **MATCH** ()-/**SHORTEST** p:~friend* COST x/-()

Disjunct shortest path queries

- **MATCH** ()-/5 **DISJUNCT SHORTEST** p:~friend*/-() (5 disjunct shortest path)

All paths : should only end in PATH PROJECT which returns the induced graph:

- **MATCH** ()-/**ALLPATHS** p:~friend* COST x/-()
  PATH PROJECT p

CONSTRUCT GRAPH ($a)-[:knows]->($b)

CONSTRUCT <fullGraphPattern> **SET** … **REMOVE**

introduce copy patterns like (=n) or ()-[=e]->()

- This copies all labels and properties
- Additionally given literal labels and properties in copy patterns are always additive

- Further before the in-line patterns:
  - **SET** n.prop = value
  - **SET** labels(n) = labels(m)
  - **SET** properties(n) = properties(m)
  - **SET** n:Label
  - **REMOVE** n.prop
  - **REMOVE** n:Label

Can only describe changes on unbound variables

Full Query: Union/Intersection

consistency condition.

- Two property graphs $G_1 = (N_1, E_1, P_1, \rho_1, \delta_1, \lambda_1, \sigma_1)$ and $G_2 = (N_2, E_2, P_2, \rho_2, \delta_2, \lambda_2, \sigma_2)$ are said to be consistent if:
- For every $e \in E_1 \cap E_2$, it holds that $\rho_1(e) = \rho_2(e)$
- For every $p \in P_1 \cap P_2$, it holds that $\delta_1(p) = \delta_2(p)$
- For every $o \in ((N_1 \cap N_2) \cup (E_1 \cap E_2) \cup (P_1 \cap P_2))$, it holds that $\lambda_1(o) = \lambda_2(o)$
- For every $o \in ((N_1 \cap N_2) \cup (E_1 \cap E_2) \cup (P_1 \cap P_2))$ and $p \in \mathbf{K}$, it holds that $\sigma_1(o,p) = \sigma_2(o,p)$

Set operations.

- Let $G_1 = (N_1, E_1, P_1, \rho_1, \delta_1, \lambda_1, \sigma_1)$ and $G_2 = (N_2, E_2, P_2, \rho_2, \delta_2, \lambda_2, \sigma_2)$ be consistent property graphs. Then, for $* \in \{\cup, \cap\}$, $G_1 * G_2$ is defined to be the property graph $G = (N, E, P, \rho, \delta, \lambda, \sigma)$ where
- $N = N_1 * N_2$, $E = E_1 * E_2$ $P = P_1 * P_2$, $\rho = \rho_1 * \rho_2$, $\delta = \delta_1 * \delta_2$, $\lambda = \lambda_1 * \lambda_2$, $\sigma = \sigma_1 * \sigma_2$
- and $*$ is the standard set-theoretic operation.

Full Query: Union/Intersection

consistency condition.

- Two property graphs $G_1 = (N_1, E_1, P_1, \rho_1, \delta_1, \lambda_1, \sigma_1)$ and $G_2 = (N_2, E_2, P_2, \rho_2, \delta_2, \lambda_2, \sigma_2)$ are said to be consistent if:
- For every $e \in E_1 \cap E_2$, it holds that $\rho_1(e) = \rho_2(e)$
- For every $p \in P_1 \cap P_2$, it holds that $\delta_1(p) = \delta_2(p)$
- For every $o \in ((N_1 \cap N_2) \cup (E_1 \cap E_2) \cup (P_1 \cap P_2))$, it holds that $\lambda_1(o) = \lambda_2(o)$
- For every $o \in ((N_1 \cap N_2) \cup (E_1 \cap E_2) \cup (P_1 \cap P_2))$ and $p \in \mathbf{K}$, it holds that $\sigma_1(o,p) = \sigma_2(o,p)$

CONSTRUCT GRAPH ($a)-[:knows]->($b)

- **Anonymous variables**.. Object patterns with no variable assigned by the user get an anonymous variable assigned.
- **Bound and unbound patterns**. All variables that occur as columns in the binding table are bound variables. All others are unbound variables. Their respective object patterns are called unbound patterns.
  - Edge patterns are only allowed to have a bound variable if they link node patterns with the same variables as in the match pattern that bound them **Projection result**. The pattern is instantiated on every tuple of binding table. Each instantiation forms a small graph isomorphic to the projection pattern. The result of the projection is the union of all pattern instantiations .
- **Pattern instantiation**.
  - for each bound object pattern: the object bound to that variable in the binding tuple.
  - for each unbound object pattern the instantiation contains a newly created object of the respective kind.

Aggregating Graph Projection

Example: ($a **GROUP** $x)-[**GROUP** $w, $y]->(**GROUP** $z)

Node pattern ($a **GROUP** $x) has grouping variable $x assigned,
Edge pattern -[**GROUP** $y]-> has grouping variable $w and $y assigned
Node pattern (GROUP $z) has grouping variable $z assigned.

- All grouping variable have to be bound variables in the binding tables.
- Bound object patterns are implicitly grouped by their own bound variable.
- Unbound node patterns not specifying grouping variables are implicitly grouped by all variables bound in the binding tables.
- Unbound edge patterns are implicitly grouped by the grouping variables of their adjacent nodes.
  - Additionally to these, unbound edge patterns may specify more grouping variables.

G-Core Examples

Co-publications weighted by venue

```
GRAPH CONSTRUCT (a)-[:COAUTHOR{weight:cost(copath)}]-(b)
PATH coauthor AS ()<-[:AUTHOR]-(paper)-[:AUTHOR]->(),
   (paper)-[p:PUBLISHED_IN]->(venue)
   WHERE venue.name IN [ "VLDB", "SIGMOD", "ICDE" ]
   COST min( case venue.type
         when “conference” then 2
         when “journal” then 1
         else infinity
         end * case p.type
         when “proceedings” then 4 else 1 end )
MATCH (a{name:"Claudio Gutierrez"}), (b{name:"Kevin Bacon"}),
    SHORTEST (a)-/copath:~coauthor*/-(b)
```

G-Core Examples

```
PATH conf_publ AS ()-[p:PUBLISHED_IN]->(:conference)
WHERE NOT p.type = "proceedings"
PATH conf_coauthor = ()<-[:AUTHOR]-(p)-[:AUTHOR]->()
WHERE (p)-/~conf_publ/-()
MATCH (a{name:”Marcelo Arenas”}), (b{name:”Paul Erdös”}),
        DISJOINT 10 SHORTEST (a)-/x:~conf_coauthor*/-(b)
CONSTRUCT GRAPH (a)-[:COAUTHOR{length:length(x)}]->(b)
```

Summary

- Closed Query Language, that allow subqueries, views
- property graphs = “directed, labeled graphs with paths, and properties on all of these”
- Powerful Language Features
  - PATH … → specifies path patterns, (weighted) shortest path finding types
    - Also ALLPATHS, using PROJECT GRAPH to return all matching edges as a graph
  - MATCH … → graph matching resulting in bindings
  - WHERE … → specifies filtering
  - CONSTRUCT GRAPH … → project bindings into a graph (incl grouping)
  - Full query: e.g. UNION of multiple CONSTRUCT GRAPH returning one graph
  - extension: CONSTRUCT TABLE … → project bindings into a table

Construct Table

CONSTRUCT TABLE *returnItems*

where *returnItems* is

- a comma-separated list of of variables bound to graph objects (nodes, edges, or paths), or
- expression followed by the keyword AS and the name of an unbound variable

Example:

- **CONSTRUCT TABLE** a, r.weight, r.weight*2 **AS** doubleWeight,
  b.name **AS** name

  MATCH (a)-[r]->(b)

  multi-valued properties are dealt with by cartesian product (or by inline aggregation, reducing them to single values)