ORACLE®

PGQL – Status Update

And Comparison to LDBC’s Graph QL proposals

Oskar van Rest
Oracle
February 9, 2017

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Program Agenda

1 Introduction to PGQL

2 What’s New in PGQL 1.0 since PGQL 0.9?

3 PGQL and LDBC’s Graph QL proposals

4 Future directions

Program Agenda

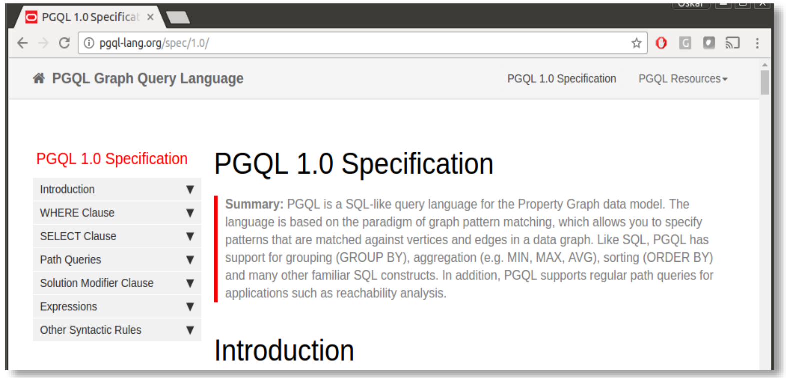
1. Introduction to PGQL
2. What’s New in PGQL 1.0 since PGQL 0.9?
3. PGQL and LDBC’s Graph QL proposals
4. Future directions

Introduction to PGQL

PGQL Graph Query Language - Overview

- Core Features
  - SQL alignment
    - **SELECT .. FROM .. WHERE ..**
    - Grouping and aggregation: **GROUP BY, AVG, MIN, MAX, SUM**
    - Solution modifiers: **ORDER BY, LIMIT, OFFSET**
  - Graph pattern matching
    - Define a high-level pattern, find all instances
    - This corresponds to basic SQL
  - (Recursive) path queries
    - Can I reach from vertex A to vertex B via some number of edges?
    - Use cases: detecting circular cash flow (fraud detection), network impact analysis, etc.

- Specification available online



PGQL 1.0 Specification

- Introduction ▼
- WHERE Clause ▼
- SELECT Clause ▼
- Path Queries ▼
- Solution Modifier Clause ▼
- Expressions ▼
- Other Syntactic Rules ▼

PGQL 1.0 Specification

> **Summary:** PGQL is a SQL-like query language for the Property Graph data model. The language is based on the paradigm of graph pattern matching, which allows you to specify patterns that are matched against vertices and edges in a data graph. Like SQL, PGQL has support for grouping (GROUP BY), aggregation (e.g. MIN, MAX, AVG), sorting (ORDER BY) and many other familiar SQL constructs. In addition, PGQL supports regular path queries for applications such as reachability analysis.

Introduction

- Implementation (PGQL 1.0)
  - Parallel Graph Analytics (PGX) ORACLE®
    - PGX is Oracle's in-memory graph analytics engine
      http://www.oracle.com/technetwork/oracle-labs/parallel-graph-analytics
    - Component of Oracle Big Data Spatial and Graph
      http://www.oracle.com/technetwork/database/database-technologies/bigdata-spatialandgraph
  - Open-sourced PGQL front-end (Apache 2.0 License)
    https://github.com/oracle/pgql-lang

ORACLE®

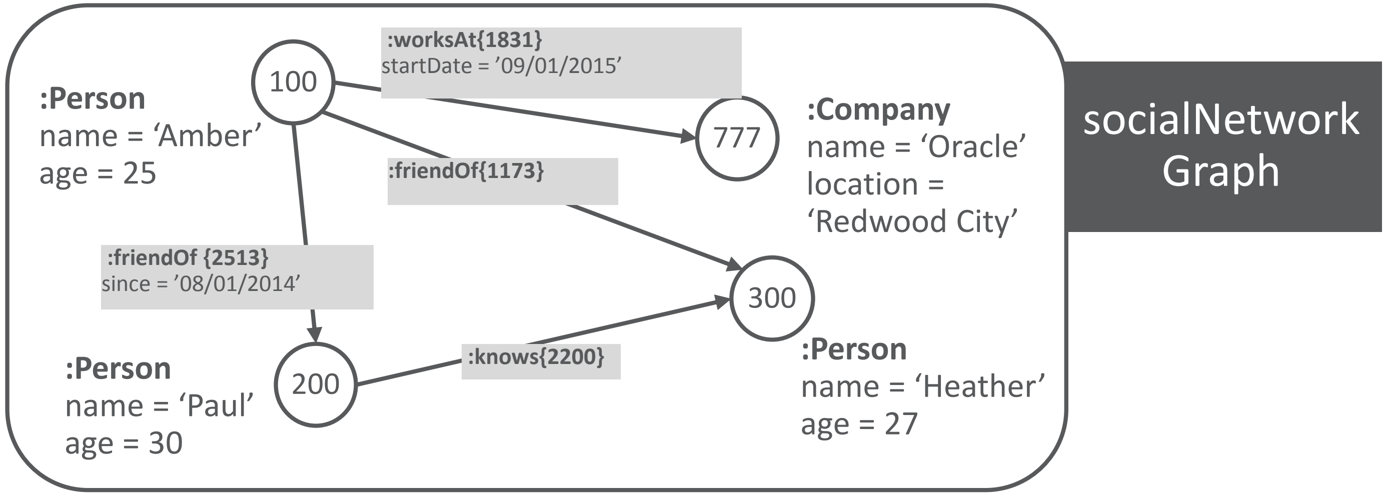
PGQL 1.0

Example query

- Find all instances of a given pattern/template in the data graph

```
SELECT v3.name, v3.age
FROM socialNetworkGraph
WHERE
  (v1:Person WITH name = ‘Amber’) -[:friendOf]-> (v2:Person) -[:knows]-> (v3:Person)
```

query



Query: Find all people who are known by friends of ‘Amber’.

https://github.com/oracle/pgql-lang/

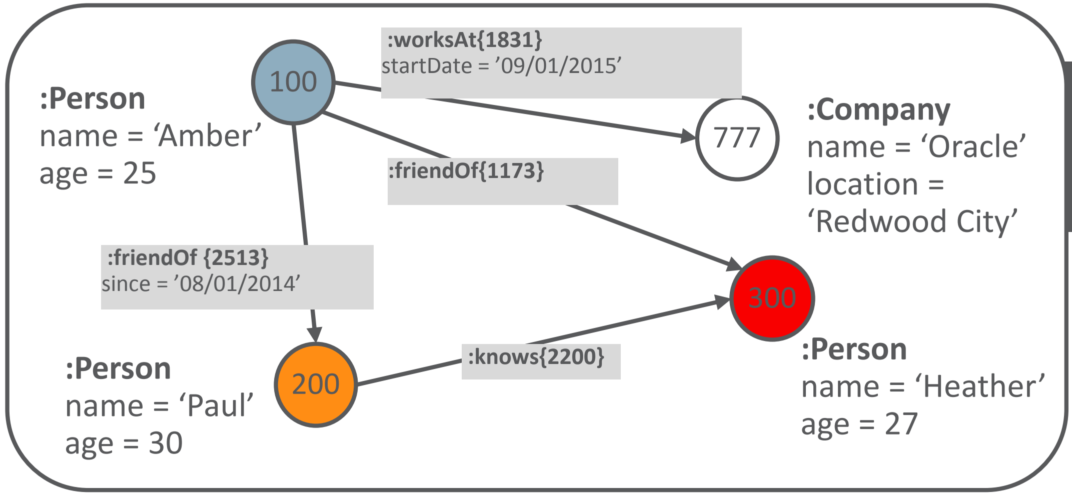
http://pgql-lang.org/

PGQL 1.0
Example query

- Find all instances of a given pattern/template in the data graph

```
SELECT v3.name, v3.age
FROM socialNetworkGraph
WHERE
  (v1:Person WITH name = ‘Amber’) -[:friendOf]-> (v2:Person) -[:knows]-> (v3:Person)
```

query



socialNetwork Graph

Query: Find all people who are known by friends of ‘Amber’.

https://github.com/oracle/pgql-lang/

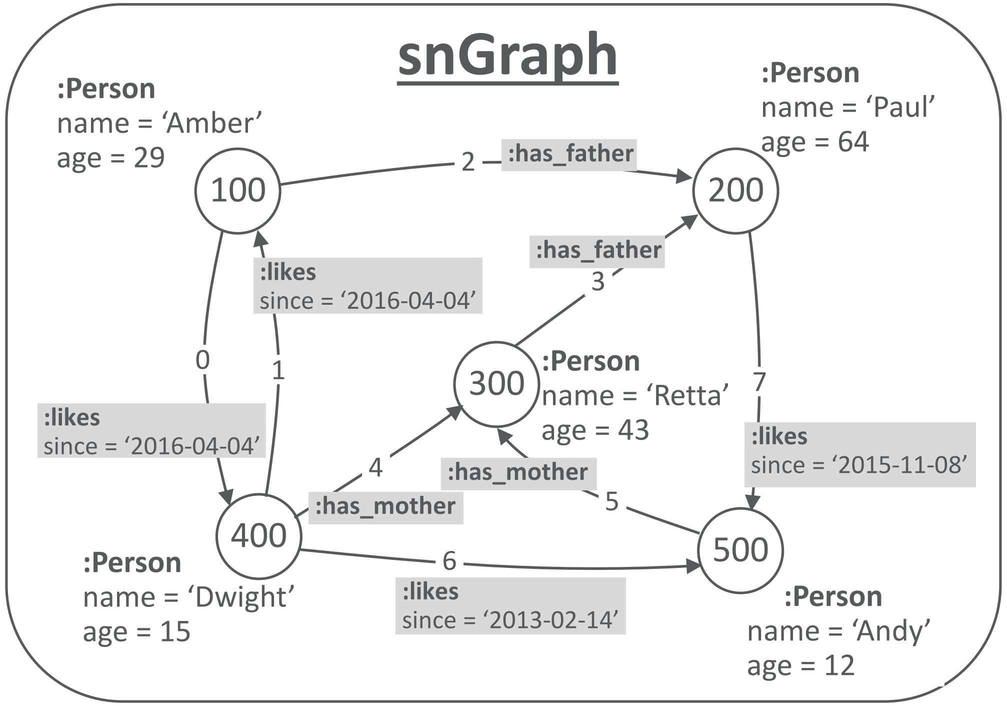
http://pgql-lang.org/

PGQL 1.0

Regular Path Queries (RPQs)

- Matching a pattern repeatedly
  - Define a **PATH** pattern at the top of a query
  - Refer to it in the WHERE clause (pattern composition)
  - Use Kleene star (*) for **repeated** matching

```
PATH has_parent := (child) -[:has_father|has_mother]-> (parent)
SELECT x.id(), y.id(), ancestor.id()
WHERE
  (x:Person WITH name = 'Andy') -/:has_parent*/-> (ancestor),
  (y) -/:has_parent*/-> (ancestor),
  x != ancestor AND y != ancestor AND x != y
```



PGQL 1.0

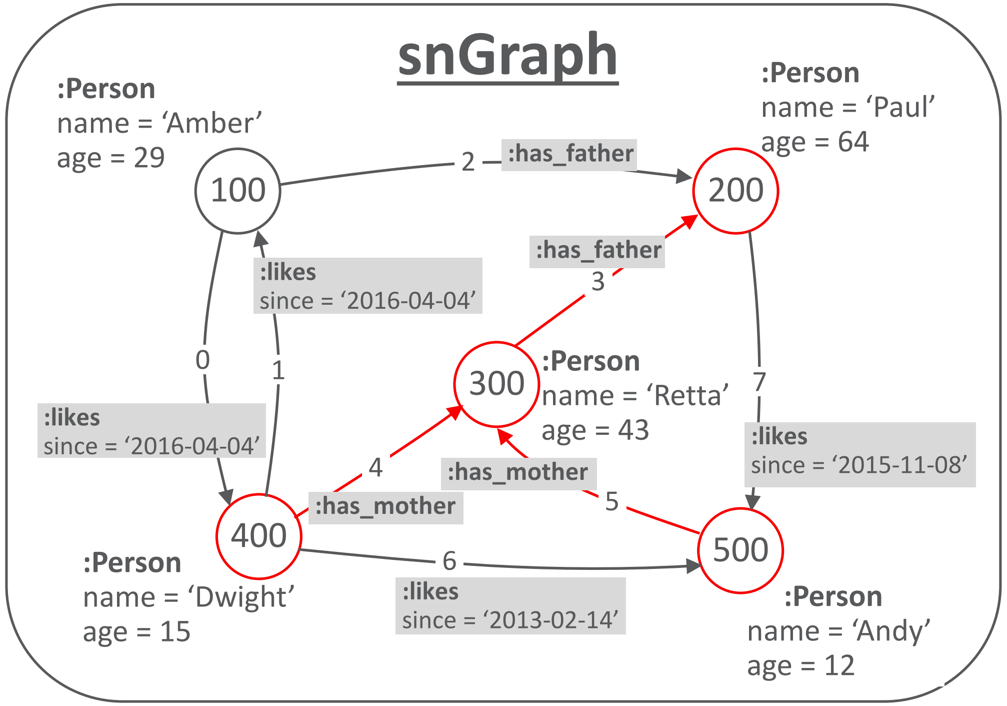
Regular Path Queries (RPQs)

- Matching a pattern repeatedly
  - Define a **PATH** pattern at the top of a query
  - Refer to it in the WHERE clause (pattern composition)
  - Use Kleene star (*) for **repeated** matching

```
PATH has_parent := (child) -[:has_father|has_mother]-> (parent)
SELECT x.id(), y.id(), ancestor.id()
WHERE
  (x:Person WITH name = 'Andy') -/:has_parent*/-> (ancestor),
  (y) -/:has_parent*/-> (ancestor),
  x != ancestor AND y != ancestor AND x != y
```

Result set

x.id()	y.id()	ancestor.id()
500	300	200
500	400	200
500	400	300



PGQL 1.0

Example: Network Impact Analysis

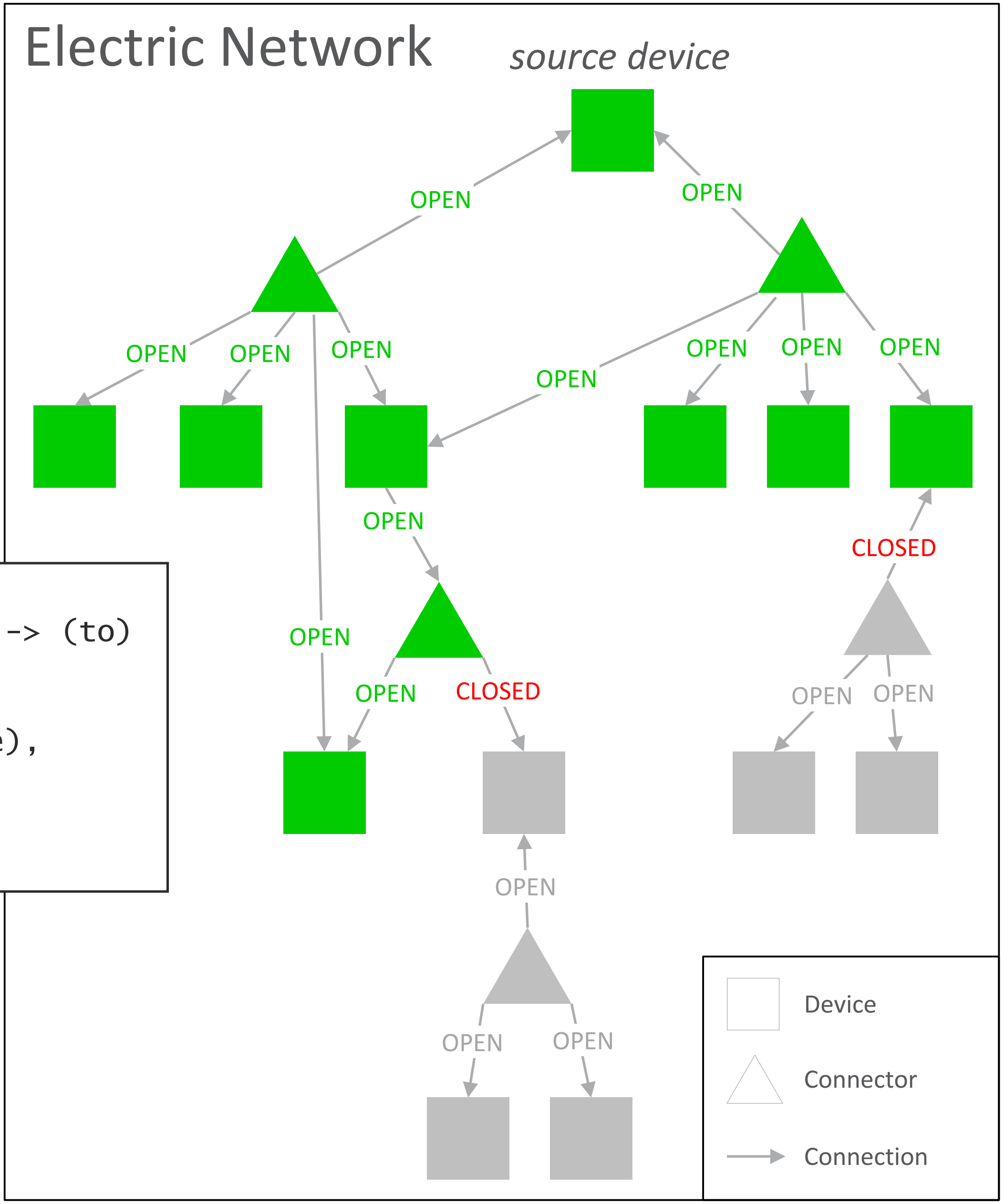
- How does a network disruption impacts reachability between devices?

Query: For each ‘Regulator’ device, show number of reachable devices following only ‘OPEN’ connections.

```
PATH connects_to :=
  (from) <-[WITH status = 'OPEN']- (connector) -[WITH status = 'OPEN']-> (to)
SELECT n.nickname, COUNT(m)
WHERE
  (n:Device WITH nickname =~ 'Regulator') -/:connects_to*/-> (m:Device),
  n != m
GROUP BY n
ORDER BY COUNT(m) DESC, n.nickname
```

Result

```
-------------------------------------------
| n.nickname                  | COUNT(m)  |
-------------------------------------------
| "Regulator, VREG2_A"        | 1596      |
| "Regulator, VREG4_B"        | 1537      |
| "Regulator, VREG4_C"        | 1537      |
| "Regulator, HVMV_Sub_RegA"  | 3         |
| "Regulator, HVMV_Sub_RegB"  | 3         |
-------------------------------------------
```



PGQL 1.0

Regular Path Queries

Comparison to SQL

Query:

Which devices are connected **transitively** to device 'Regulator, HVMV_Sub_RegB‘?

PGQL

```
PATH connects_to := (from) <- (connector) -> (to)
SELECT y.name
WHERE (x:Device) -/:connects_to*/-> (y:Device),
      x.name = 'Regulator, HVMV_Sub_RegB'),
      x != y
```

SQL

```
WITH temp(device_id, device_name) AS (
  -- Anchor member:
  SELECT device_id, name
  FROM   Devices
  WHERE  name = 'Regulator, HVMV_Sub_RegB‘
UNION ALL
  -- Recursive member:
  SELECT Devices.device_id, Devices.name
  FROM   temp, Devices, Connections conn1,
         Connections conn2, Connectors
  WHERE  temp.device_id = conn1.to_device_id
    AND  conn1.from_connector_id = Connectors.connector_id
    AND  Connectors.connector_id = conn2.from_connector_id
    AND  conn2.to_device_id = Devices.device_id
    AND  temp.device_id != Devices.device_id)
CYCLE device_id SET cycle TO 1 DEFAULT 0
SELECT DISTINCT device_name
FROM temp
WHERE cycle = 0
  AND device_name != 'Regulator, HVMV_Sub_RegB'
```

Program Agenda

1 Introduction to PGQL

2 What's New in PGQL 1.0 since PGQL 0.9?

3 PGQL and LDBC's Graph QL proposals

4 Future directions

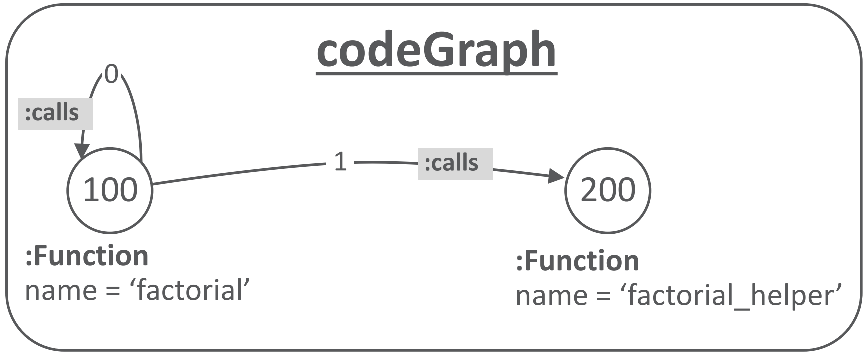
What’s New in PGQL 1.0 since PGQL 0.9?

What’s New in PGQL 1.0 since PGQL 0.9?

- Regular Path Queries (RPQs) (see previous slides)
  - PGQL currently supports *reachability* RPQs only
  - Future versions will have min-hop/weighted *shortest path finding* RPQs
- Changed pattern matching semantic: isomorphism => homomorphism
  - Isomorphism has the restriction that two query vertices should not map to the same data vertex

```
SELECT f2.name
WHERE (f1:Function WITH name = ‘factorial’) -[:calls]-> (f2)
```

Query: “which functions are called by function ‘factorial’?”



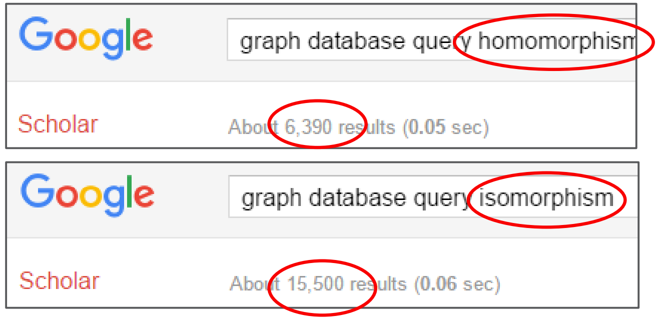
Result with **isomorphism**

f2.name
‘factorialHelper’

Result with **homomorphism**

f2.name
‘factorial’
‘factorialHelper’

PGQL 0.9 => PGQL 1.0
Isomorphism => homomorphism



graph database query homomorphism

About 6,390 results (0.05 sec)

graph database query isomorphism

About 15,500 results (0.06 sec)

According to several publications, graph querying comes down to subgraph isomorphism, but this is not always the case.

- Isomorphism semantic found to be more intuitive for first-time users
  - *(not based on empirical study)*
  - Homomorphism may return more results than expected (e.g. "find friends of friends of 'John'" returns 'John')
- Isomorphism has limitations (see previous slide)
- Both have the same worst-case time complexity: $O(n^k)$ (**n** = num. data vertices, **k** = num. query vertices)
  - However, if we apply isomorphism to recursive path queries, things blow up
- Also, isomorphism doesn't translate well to/**from** SQL, but homomorphism does
- Hence, PGQL is now based on homomorphism
  - We also plan to introduce an **allDifferent(v1, v2, …)** function to avoid large numbers of non-equality constraints: **allDifferent(x, y, z)** instead of **x != y, x != z, y != z**

Program Agenda

1. Introduction to PGQL
2. What’s New in PGQL 1.0 since PGQL 0.9?
3. PGQL and LDBC’s Graph QL proposals
4. Future directions

ORACLE

PGQL and LDBC’s Graph QL proposals

Pattern matching in the FROM clause

```
SELECT *
WHERE
  (x:Person WITH name = ‘Ann’) -[e:likes]-> (y),
  x.age = y.age
```

```
SELECT *
FROM (x:Person) -[e:likes]-> (y)
WHERE
  x.name = ‘Ann’ AND x.age = y.age
```

SPARQL-like

SQL-like

- Idea came from other task force members
  - Aligns better with SQL
    - Labels ‘Person’ and ‘likes’ correspond to table names in SQL’s FROM clause
    - WHERE clause only contains filters like in SQL and no graph pattern

- Disadvantage is that negation of graph patterns is not so concise:

```
SELECT *
WHERE
  (x) -> (y),
  NOT EXISTS { (x) <- (y) }
```

```
SELECT *
FROM (x) -> (y)
WHERE
  NOT EXISTS (
    SELECT *
    FROM (x) <- (y)
  )
```

Query: “find all edges that don’t have a reverse edge”

Path queries: **comparing data along paths**

Regular Expressions with Memory (REM) [1]

- REMs are Regular Path Queries (RPQs) with registers to store properties of vertices/edges along paths
  - Stored properties can be used later on during traversal to compare against other properties
- Most expressive (powerful) RPQ formalism with same complexity as usual RPQs
- Hard to come up with a syntax for REMs that is declarative

[1] http://homepages.inf.ed.ac.uk/s1058408/data/jcss.pdf

Idea proposed for PGQL / Graph QL

- PATH patterns with WHERE clause for data comparison

Query: “find devices that are reachable from ‘power_generator_x29’ via a path such that all the devices along the path have equal voltage”

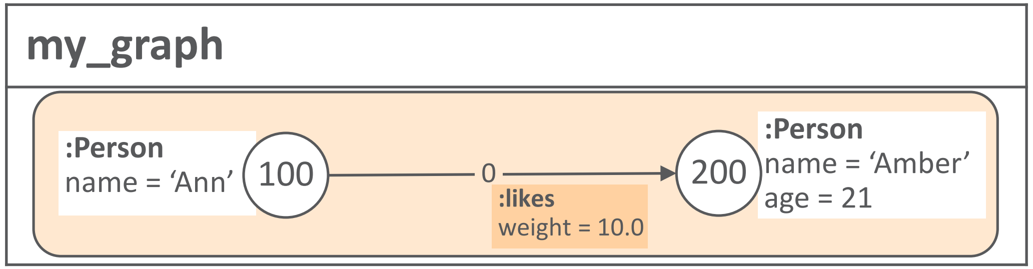
```
PATH eq_voltage_hop:=
  (n:Device) -> (m:Device)
  WHERE n.voltage = m.voltage
SELECT y.name
FROM (x) -/:eq_voltage_hop+/-> (y)
WHERE x.name = ‘power_generator_x29’
```

- Supports a subset of REM, but is declarative
- Paths can be processed in either direction
  (either from *x* to *y* or from *y* to *x*)

Recent proposals from LDBC’s Graph QL work force

Graph QL proposal #1

- Unified data model: tables with cells that hold graphs
  - Cells may also hold paths, vertices, edges, Strings, Integers, etc.



Graph QL proposal #2

- Unified data model: graphs encoded as two tables
  - One row per vertex/edge

my_graph_vT

v_id	name	age
100	‘Ann’	NULL
200	‘Amber’	21

my_graph_eT

e_id	v1_id	v2_id	weight
0	100	200	10.0

  - Still figuring out how to encode paths

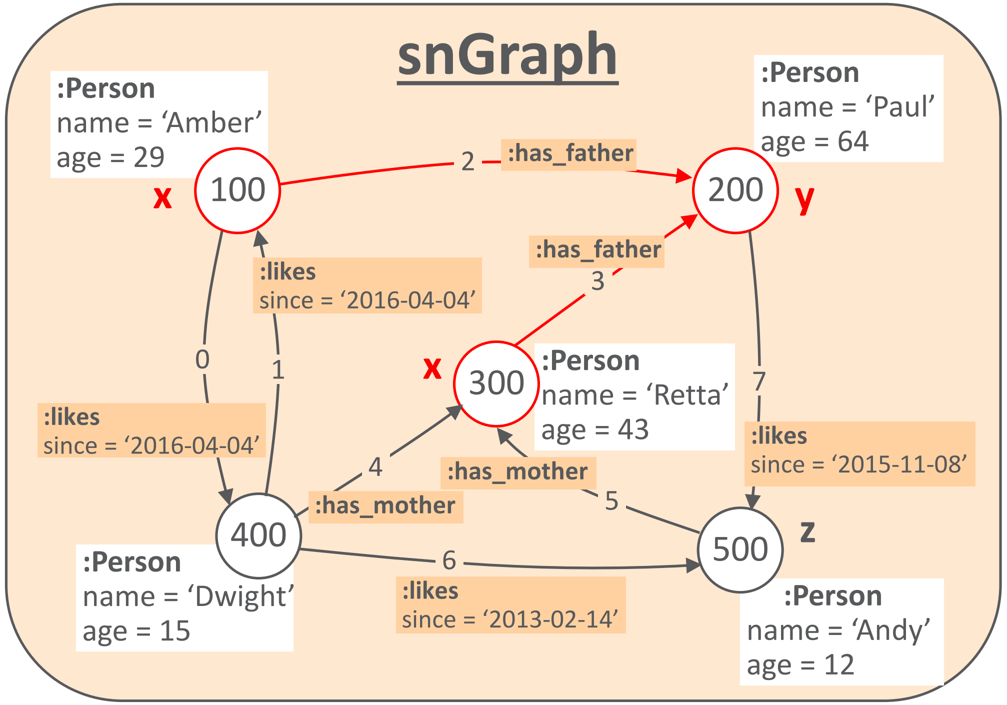
This is like **PGQL**

- i.e. **tables** with complex data types as **output**
- but… PGQL has **graphs** (instead of tables) as **input**
  - Seems practical
  - But ***not* a unified data model**

Not part of PGQL 1.0

Graph Construction in PGQL

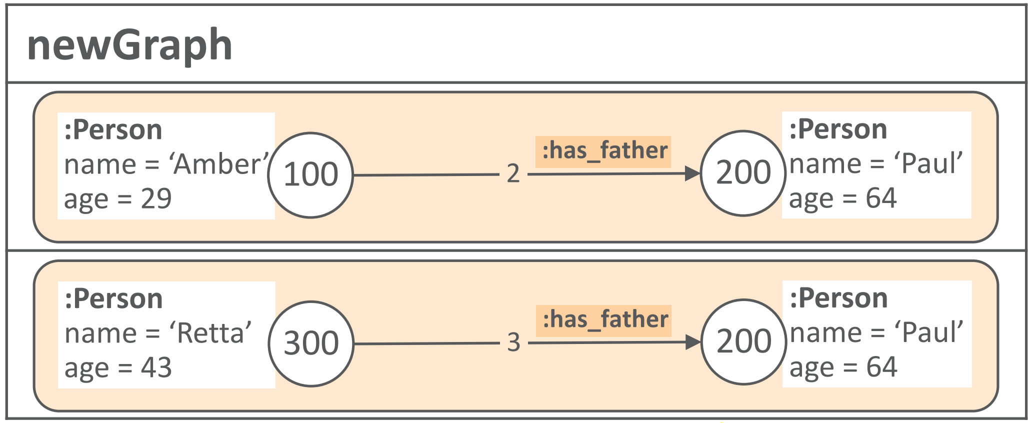
- Specify graph production pattern in SELECT
  - Pattern may contain *existing* vertices / edges / paths
  - Pattern may contain *new* vertices / edges / properties (not shown here)



Query: “construct new graphs, each containing one ‘has_father’ edge from the input graph

```
SELECT { (x) -[e]-> (y) } AS newGraph
FROM (x) -[e:has_father]-> (y) IN GRAPH snGraph
```

newGraph



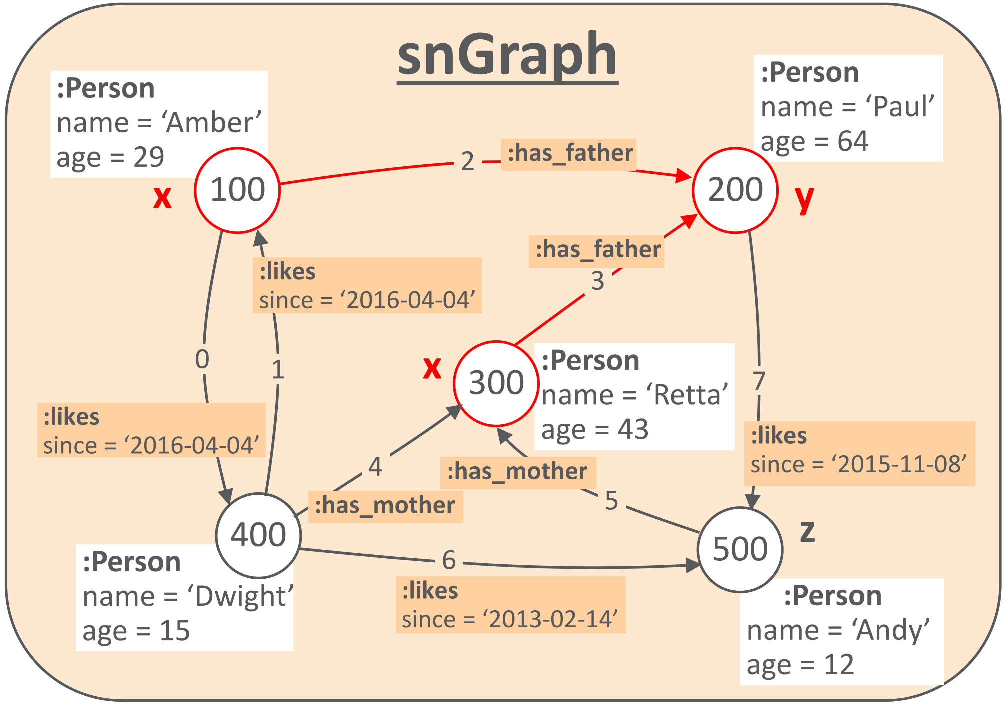
Query result: table with graphs

Not part of PGQL 1.0

New aggregate: FUSION (may be used in combination with GROUP BY)

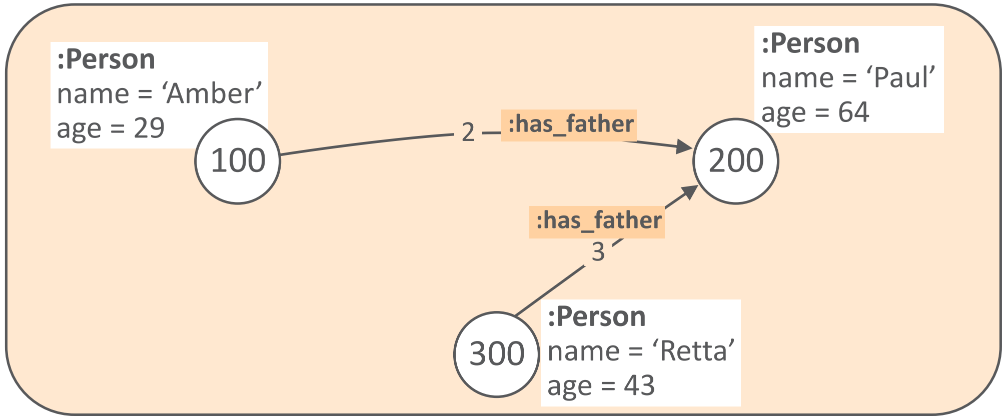
- The FUSION aggregate merges a set of graphs into a single (large) graph

Query: “construct a new graph containing all the ‘has_father’ edges from the input graph



```
SELECT FUSION({ (x) -[e]-> (y) } ) AS newGraph
FROM (x) -[e:has_father]-> (y) IN GRAPH snGraph
```

newGraph



Query result: table with graphs

Not part of PGQL 1.0

Composition of queries that return graphs

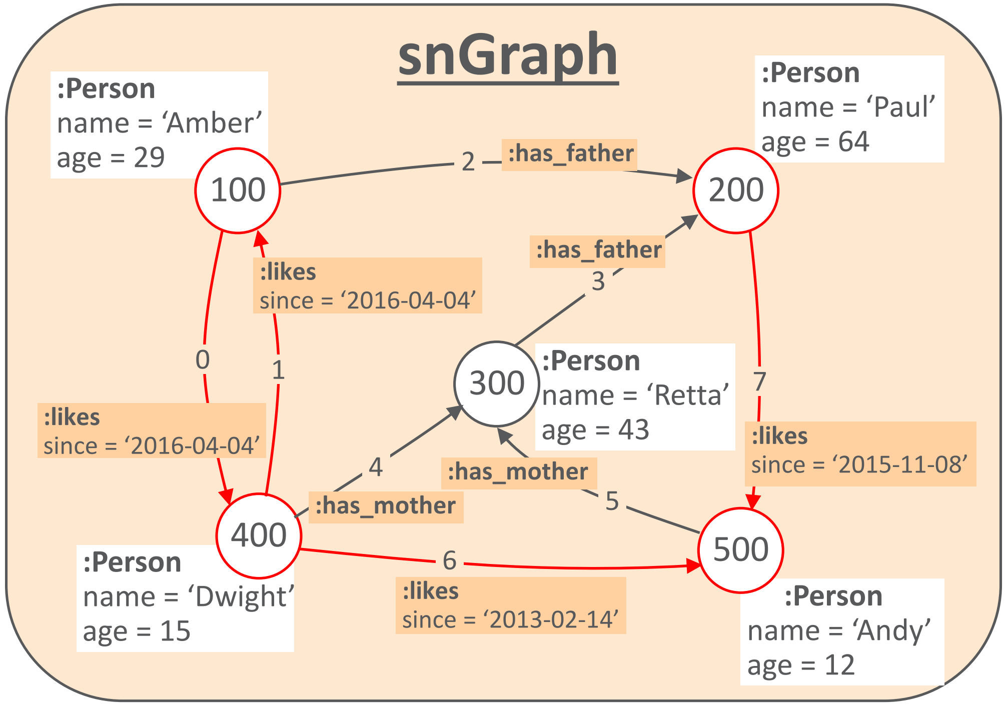
- PGQL takes a graph as input and returns a table as output (not a unified data model)
- Yet, can naturally compose queries that return graphs:

```
SELECT COUNT(*)
FROM (n) IN GRAPH (
  SELECT FUSION({(a) -[e]-> (b)})
  FROM (a) -[e:likes]-> (b) IN GRAPH snGraph
)
```

Query result:

COUNT(*)
4

Inner query: returns a graph that contains only 'likes' edges.
Outer query: returns the number of vertices in that graph.



Program Agenda

1 Introduction to PGQL

2 What's New in PGQL 1.0 since PGQL 0.9?

3 PGQL and LDBC's Graph QL proposals

4 Future directions

ORACLE

Future directions

ORACLE®

Future Directions

Querying Multiple Graphs

- Open Question: How to refer to input graphs?
  - Option 1: Refer to each graph by name (like SPARQL)
  - Option 2: Refer to a column of an input table containing an arbitrary number of graphs (like LDBC's Graph QL proposal #1)

Not really the typical use case we see

- Open Question: How to connect data from different graphs?
  - Option 1: Merge graphs together first (**fusion(g1, g2, g3, …)**), then do pattern matching (similar to SPARQL)
  - Option 2: Match different parts of the pattern on different input graphs, then join on certain properties (like LDBC's Graph QL proposal #2)

Works well for RDF graphs where vertices have UUIDs. May not work for Property Graphs.

Future Directions

SQL Extension

- Introduce ‘pattern matching queries’ in SQL

Query: “find people who follow Angela Merkel on Twitter”

```
SELECT Person.first_name
FROM Person
WHERE EXISTS (
   SELECT *
   FROM (n) -[:follows]-> (m) IN GRAPH twoTablesToGraph(twitter_vT, twitter_eT)
   WHERE Person.first_name = n.name AND m.name = ‘Angela Merkel’
)
```

Standard SQL query

Pattern matching SQL query

  - Need standard way(s) of storing graphs as tables (two options below) and a way to access such graphs in SQL (e.g. using a function **twoTablesToGraph(vT, eT)**)

Option 1: vT/eT tables with one row per vertex/edge (handles dense and structured data well)

v_id	name	age
100	‘John’	NULL
200	‘Amber’	21

Option 2: vT/eT tables with one row per property (handles sparse and unstructured data well)

v_id	prop_name	string_value	int_value
100	‘name’	‘John’	NULL
200	‘name’	‘Amber’	NULL
100	‘age’	NULL	21

Safe Harbor Statement

The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

ORACLE®