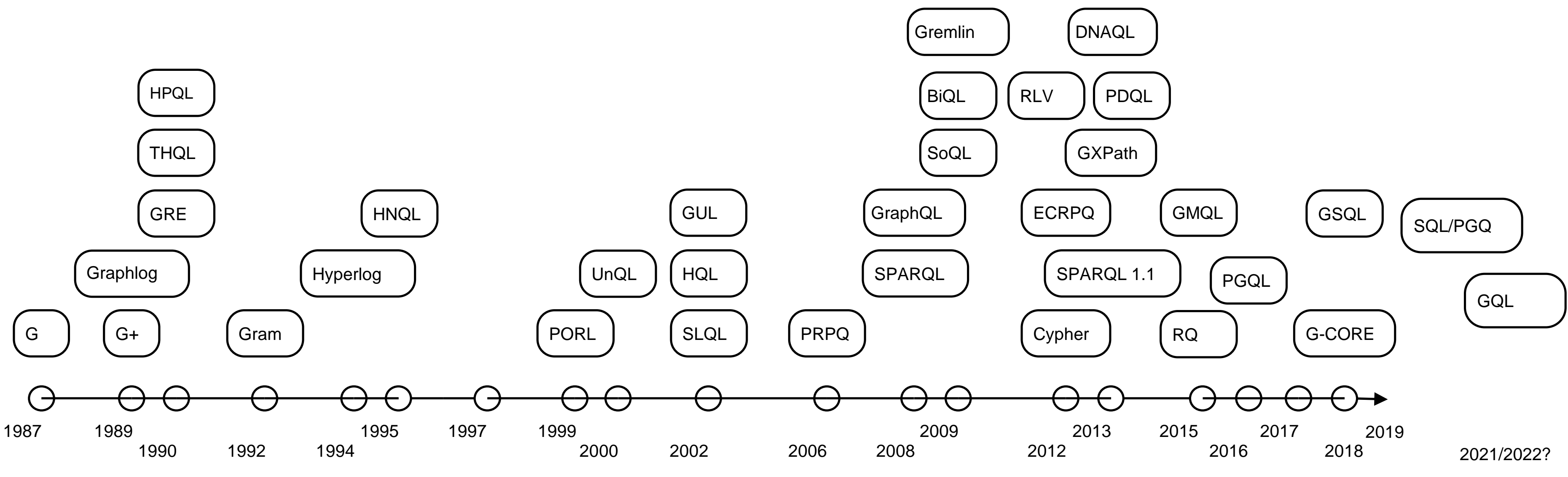
A Survey Of Current Property Graph Query Languages

Peter Boncz (CWI)

incorporating slides from:

Renzo Angles (Talca University),
Oskar van Rest (Oracle),
Mingxi Wu (TigerGraph) &
Stefan Plantikow (neo4j)

History of Graph Query Languages



SPARQL | Cypher | Gremlin | PGQL | GSQL | G-CORE | SQL/PGQ | GQL

History: the query language G

- By Isabel Cruz, Alberto Mendelzon & Peter Wood
- Data model: simple graphs
- Formal and Graphical forms
- Main functionality
  - Graph pattern queries
  - Path finding queries

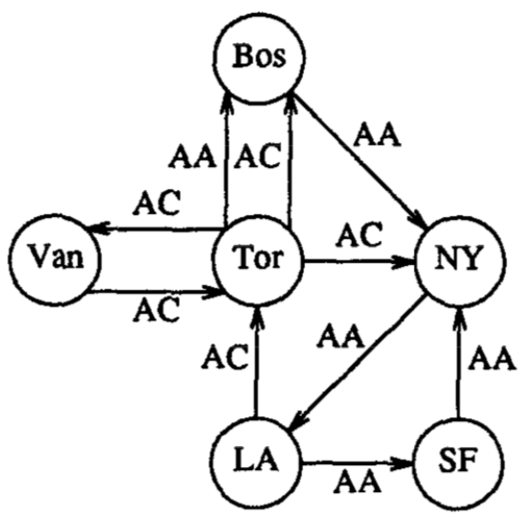
ABSTRACT

We define a language **G** for querying data represented as a labeled graph *G* By considering *G* as a relation, this graphical query language can be viewed as a relational query language, and its expressive power can be compared to that of other relational query languages We do not propose **G** as an alternative to general purpose relational query languages, but rather as a complementary language in which recursive queries are simple to formulate The user is aided in this formulation by means of a graphical interface The provision of regular expressions in **G** allows recursive queries more general than transitive closure to be posed, although the language is not as powerful as those based on function-free Horn clauses However, we hope to be able to exploit well-known graph algorithms in evaluating recursive queries efficiently, a topic which has received widespread attention recently

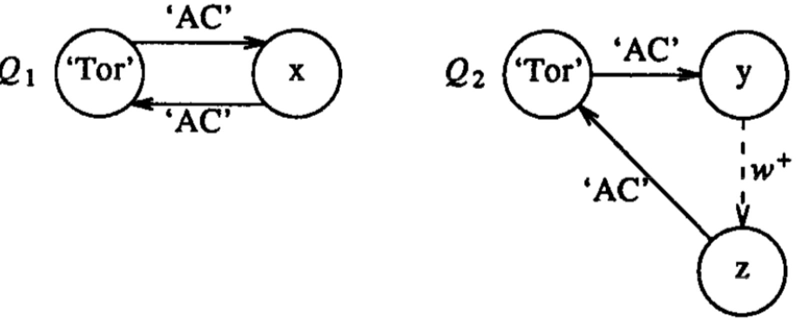
I. F. Cruz et al. A graphical query language supporting recursion. SIGMOD 1987.

G Example

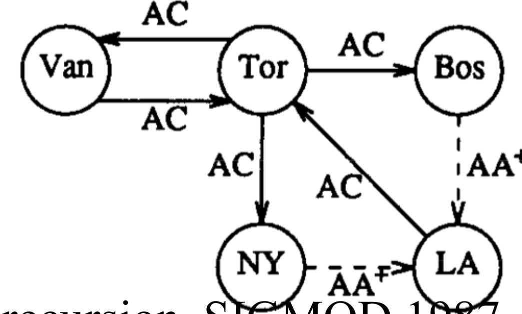
EXAMPLE 1 The following graph represents the flight information of various airlines Each node is labeled by the name of a city, while each edge is labeled by an airline name



EXAMPLE 2 Given the graph G of Example 1, the following query $Q = \{Q_1, Q_2\}$ finds the first and last cities visited in all round trips from Toronto, in which the first and last flights are with Air Canada and all other flights (if any) are with the same airline



The following graph is the value of Q with respect to the graph G



I. F. Cruz et al. A graphical query language supporting recursion. SIGMOD 1987.

Systems: Popular Query Language Implementations

SQL

- MySQL, SQLserver, Oracle, SQLserver, Postgres, Redis, DB2, Amazon Aurora, Amazon Redshift, Snowflake, Spark SQL, etc etc etc *(398000k google hits for `sql query')*

SPARQL

- Amazon Neptune, Ontotext, GraphDB, AllegroGraph, Apache Jena with ARQ, Redland, MarkLogic, Stardog, Virtuoso, Blazegraph, Oracle DB Enterprise Spatial & Graph, Cray Urika-GD, AnzoGraph *(1190k google hits for 'sparql query')*

Cypher

- neo4j, RedisGraph, neo4j CAPS (Cypher on APache Spark), SAP HANA, Agens Graph, AnzoGraph, Cypher for Gremlin, Memgraph, OrientDB *(343k google hits for 'cypher query')*

Gremlin

- Amazon Neptune, (IBM) JanusGraph (ex TitanDB), Datastax Enterprise Graph, Azure Cosmos DB, Stardog, neo4j, BlazeGraph, OrientDB, GRAKN.AI *(320k google hits for 'gremlin query')*

GSQL

- TigerGraph *(21k google hits for 'gsql query')*

PGQL

- Oracle (Big Data) Spatial and GraphOracle Labs PGX (+Oracle Labs PGX.D) *(7k google hits for 'pgql query')*

Graph Query Language Functionalities

- Graph Navigation
  - Graph Pattern Matching {homomorphic, isomorphic} into variables
    - ➔ **graph** in, (binding) **table** out
  - (Regular Pattern) Path Finding {ALL, SHORTEST, CHEAPEST}
    - ➔Q: how do paths fit the PG data model??
  - Filters (Boolean conditions on matches, existence of paths)
- Graph Construction
  - Grouping by {existing vertex/edge, value combination}
  - Merging new elements into graphs (possibly temporarily)
- Value Joins {inner,outer,anti} (possibly between multiple Graphs or even tables)
- Union/Intersection/Difference (between multiple Graphs)
- Graph Views
- Subqueries (correlated or not)
- Updates (deletion, insertion, update)

G-CORE:

A Core for Future Graph Query Languages

LDBC GraphQL task force

GCORE is the culmination of 2.5 years of intensive discussion between LDBC and **industry**, including:

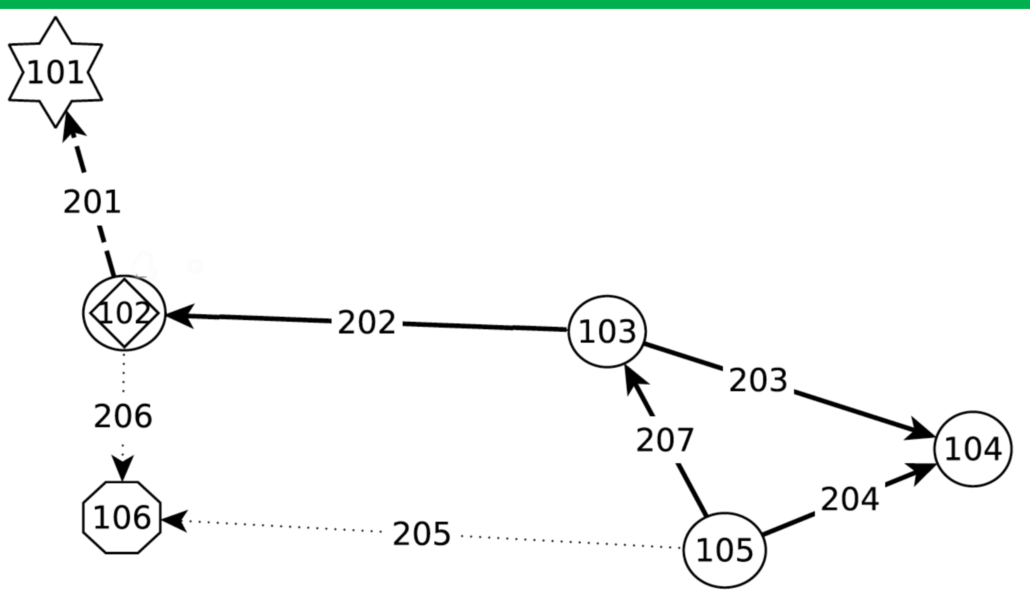
HP, Huawei, IBM, Neo4j, Oracle, SAP and Sparsity

LDBC Graph Query Language Task Force

- Recommend a query language core that will strengthen future versions of industrial graph query languages.
- Perform deep academic analysis of the expressiveness and complexity of evaluation of the query language
- Ensure a powerful yet practical query language

Academia	Industry
Renzo Angles, Universidad de Talca	Alastair Green, Neo4j
Marcelo Arenas, PUC Chile (leader)	Tobias Lindaaker, Neo4j
Pablo Barceló, Universidad de Chile	Marcus Paradies, SAP
Peter Boncz, CWI	Stefan Plantikow, Neo4j
George Fletcher, Eindhoven University of Technology	*Arnau Prat, Sparsity*
Claudio Gutierrez, Universidad de Chile	Juan Sequeda, Capsenta
Hannes Voigt, TU Dresden	Oskar van Rest, Oracle

Graph Data Model



- **directed** graph
- nodes & edges are **entities**
- entities can have **labels**

Node Labels

Person Place Tag Manager

Edge Labels

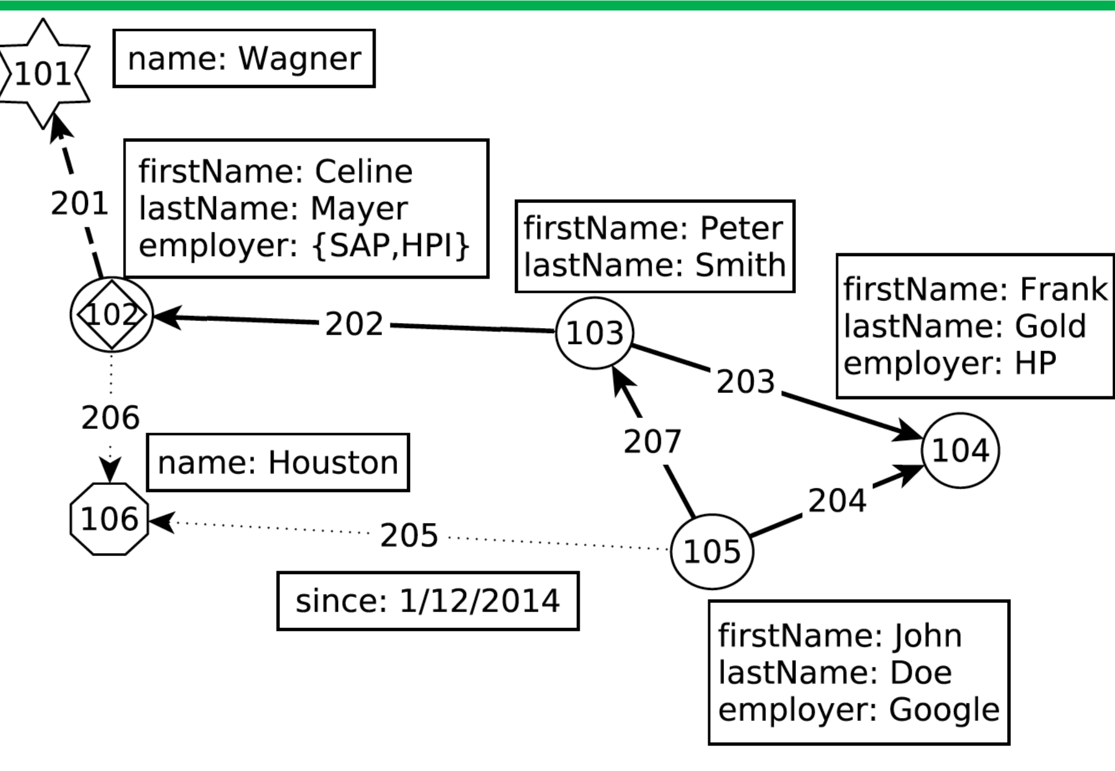
knows isLocatedIn hasInterest

Example from **SNB**:

LDBC Social Network Benchmark

(see SIGMOD 2015 paper)

Property Graph Data Model



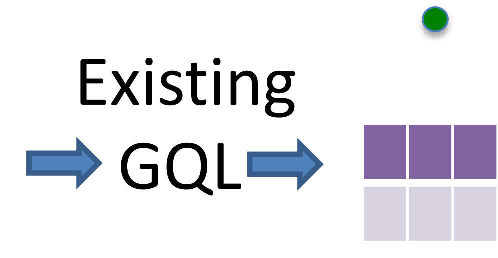
- **directed** graph
- nodes & edges are **entities**
- entities can have **labels**
- ..and (**property,value**) pairs

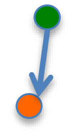
CHALLENGE 1: COMPOSABILITY

- Current graph query languages are **not** composable
  - In: Graphs
  - Out: Tables, (list of) Nodes, Edges
    - Not: **Graph**

CHALLENGE 1: COMPOSABILITY

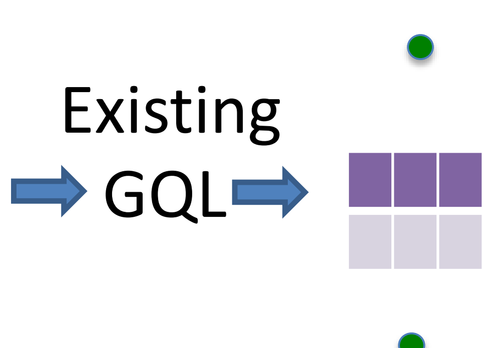
- Current graph query languages are **not** composable
  - In: Graphs
  - Out: Tables, (list of) Nodes, Edges
    - Not: **Graph**





CHALLENGE 1: COMPOSABILITY

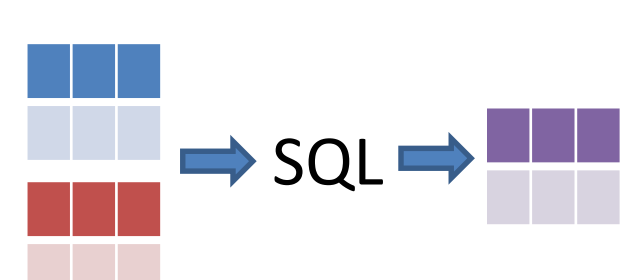
- Current graph query languages are **not** composable
  - In: Graphs
  - Out: Tables, (list of) Nodes, Edges
    - Not: **Graph**

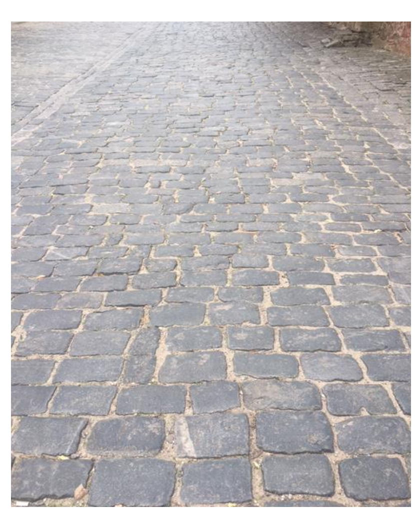


- Why is it important?
  - No Views and Sub-queries
  - Diminishes expressive power of the language

CHALLENGE 1: COMPOSABILITY

- Current graph query languages are **not** composable
  - In: Graphs
  - Out: Tables, (list of) Nodes, Edges
    - Not: **Graph**
- Why is it important?
  - No Views and Sub-queries
  - Diminishes expressive power of the language





CHALLENGE 2: PATHS

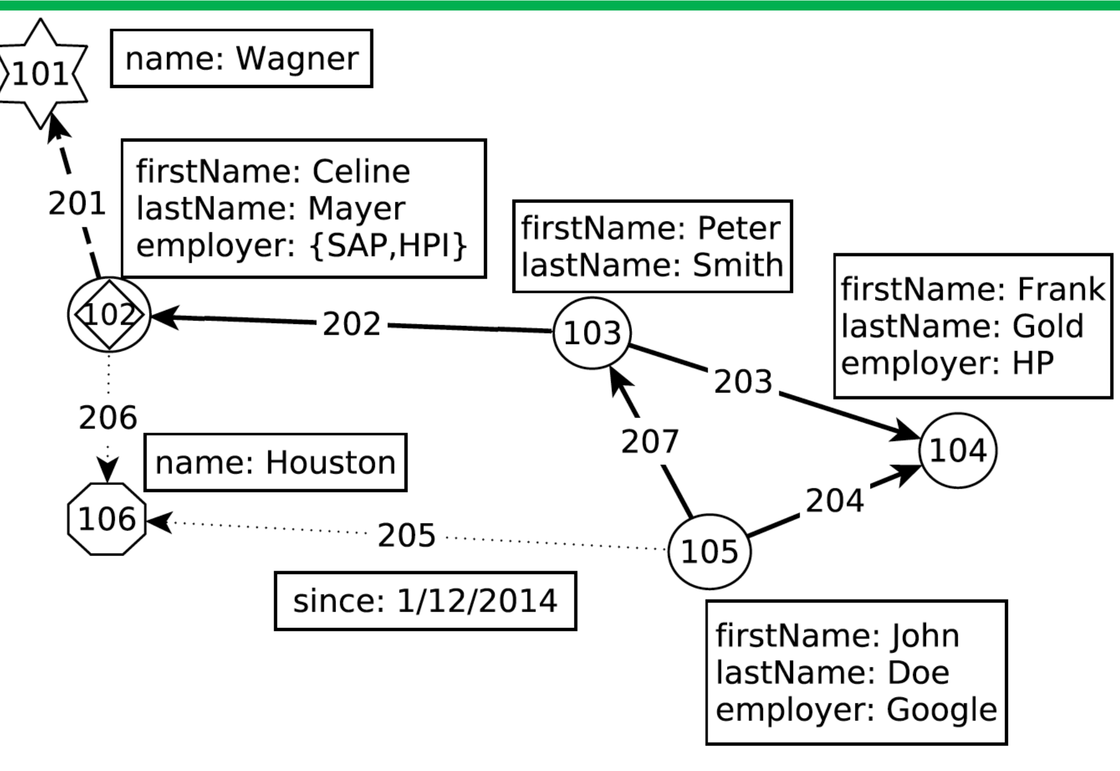
- Current graph query languages treat paths as second class citizens
  - Paths that are returned have to be post-processed in the client (a list of nodes or edges)

CHALLENGE 2: PATHS

- Current graph query languages treat paths as second class citizens
  - Paths that are returned have to be post-processed in the client (a list of nodes or edges)

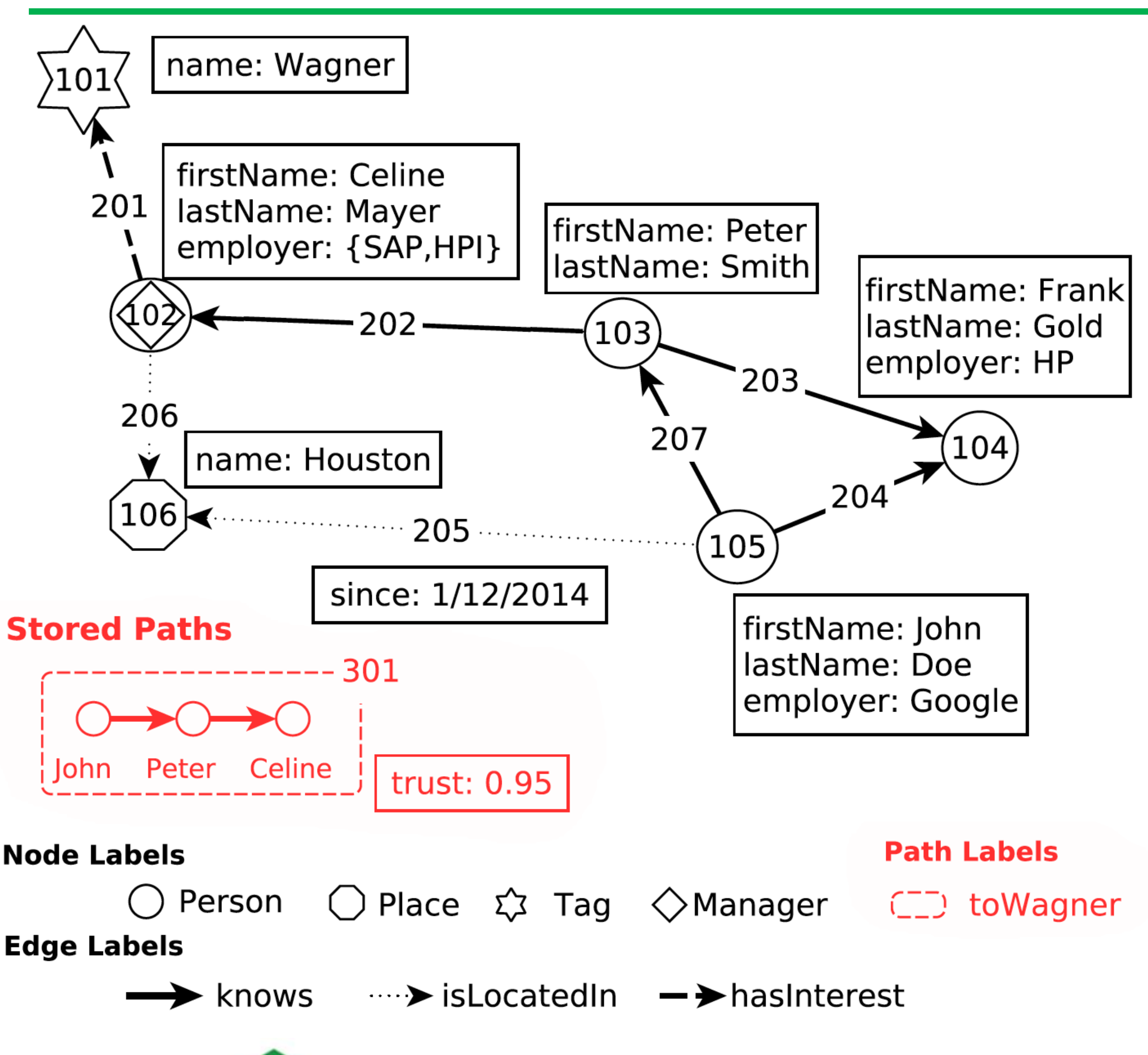
- Why is it important?
  - Paths are fundamental to Graphs
  - Increase the expressivity of the language; do more within the language

Property Graph Data Model



- **directed** graph
- nodes & edges are **entities**
- entities can have **labels**
- ..and (**property,value**) pairs

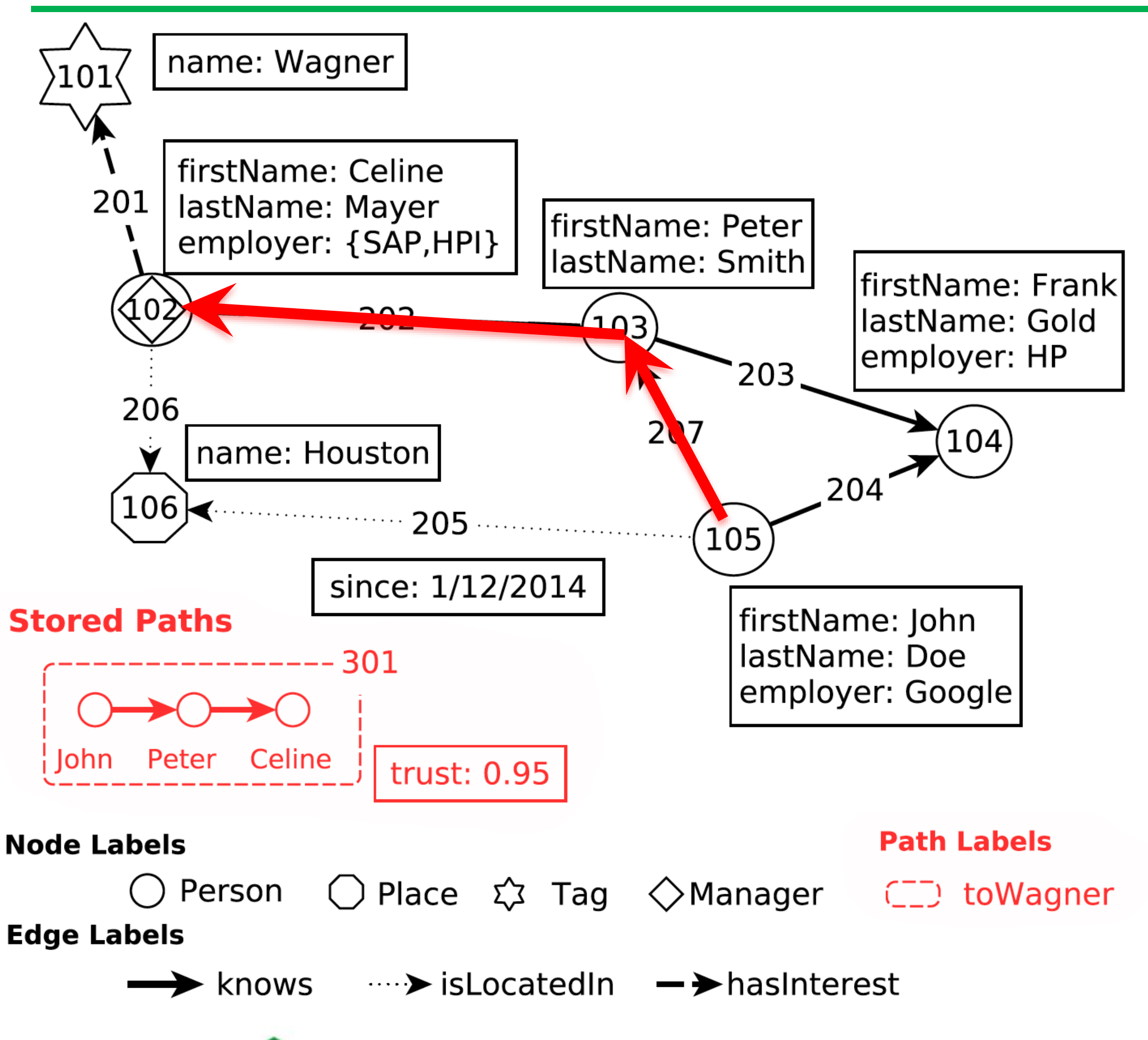
Path Property Graph Data Model



- **directed** graph
- paths, nodes & edges are **entities**
- entities can have **labels**
- ..and (**property,value**) pairs

LDBC

Path Property Graph Data Model



- **directed** graph
- paths, nodes & edges are **entities**
- entities can have **labels**
- ..and (**property,value**) pairs

a **path** is a sequence of consecutive edges in the graph

CHALLENGE 3: TRACTABILITY

- Graph query languages in handling paths can easily define functionality that is provably intractable. For instance,
  - enumerating paths,
  - returning paths without cycles (simple paths),
  - supporting arbitrary conditions on paths,
  - optional pattern matching, etc..

CHALLENGE 3: TRACTABILITY

- Graph query languages in handling paths can easily define functionality that is provably intractable. For instance,
  - enumerating paths,
  - returning paths without cycles (simple paths),
  - supporting arbitrary conditions on paths,
  - optional pattern matching, etc..
- G-CORE connects the practical work done in industrial proposals with the foundational research on graph databases
  - G-CORE is **tractable** in data complexity (=can be implemented efficiently)

Always returning a graph

```
CONSTRUCT (n)
MATCH (n:Person) ON social_graph
WHERE n.employer = ’Google'
```

- **CONSTRUCT** clause: Every query returns a graph
  - New graph with only nodes: those persons who work at Google
  - All the labels and properties that these person nodes had in social_graph are preserved in the returned result graph.

Syntax inspired by Neo4j’s Cypher and Oracle’s PGQL

Multi-Graph Queries and Joins

- Simple data integration query

```
CONSTRUCT (c)<-[:worksAt]-(n)
MATCH (c:Company) ON company_graph,
      (n:Person) ON social_graph
WHERE c.name = n.employer
UNION social_graph
```

Multi-Graph Queries and Joins

- Simple data integration query

```
CONSTRUCT (c)<-[:worksAt]-(n)
MATCH (c:Company) ON company_graph,
      (n:Person) ON social_graph
WHERE c.name = n.employer
UNION social_graph
```

- Load company nodes into company_graph
- Create a unified graph (**UNION**) where employees and companies are connected with an edge labeled worksAt.

Multi-Graph Queries and Joins

- Simple data integration query

```
CONSTRUCT (c)<-[:worksAt]-(n)
MATCH (c:Company) ON company_graph,
      (n:Person) ON social_graph
WHERE c.name = n.employer
UNION social_graph
```

c	n
0 #Google	105 #John
1 #HPI	104 #Frank
2 #SAP	102 #Celine
3 #HP	102 #Celine

- Load company nodes into company_graph
- Create a unified graph (**UNION**) where employees and companies are connected with an edge labeled worksAt.

Multi-Graph Queries and Joins

- Simple data integration query

```
CONSTRUCT (c)<-[:worksAt]-(n)
MATCH (c:Company) ON company_graph,
      (n:Person) ON social_graph
WHERE c.name = n.employer
UNION social_graph
```

c	n
0 #Google	105 #John
1 #HPI	104 #Frank
2 #SAP	102 #Celine
3 #HP	102 #Celine

- Load company nodes into company_graph
- Create a unified graph (**UNION**) where employees and companies are connected with an edge labeled worksAt.

c
0 #HPI
1 #SAP
2 #Google
3 #HP

n
105 #John
104 #Frank
103 #Peter
102 #Celine

Multi-Graph Queries and Joins

- Simple data integration query

```
CONSTRUCT (c)<-[:worksAt]-(n)
MATCH (c:Company) ON company_graph,
      (n:Person) ON social_graph
WHERE c.name = n.employer
UNION social_graph
```

- Load company nodes into company_graph
- Create a unified graph (**UNION**) where employees and companies are connected with an edge labeled worksAt.

c	n
0 #Google	105 #John
1 #HPI	104 #Frank
2 #SAP	102 #Celine
3 #HP	102 #Celine

$\sigma_{c.name=n.employer}$

$\times$

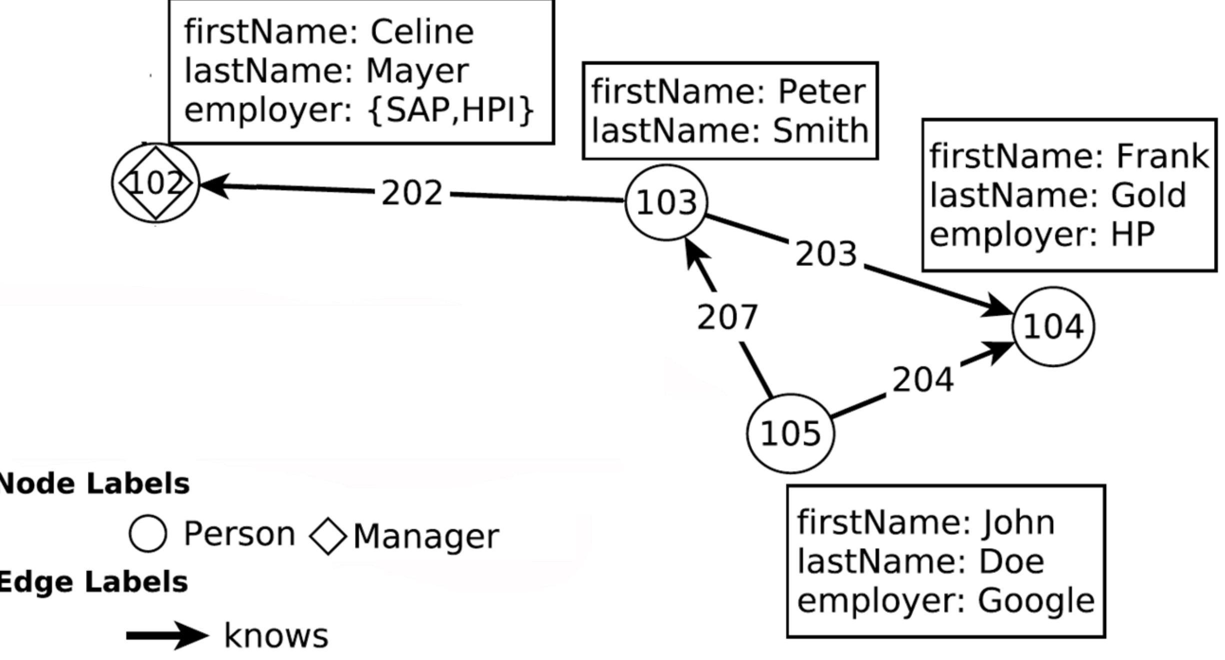
c
0 #HPI
1 #SAP
2 #Google
3 #HP

n
105 #John
104 #Frank
103 #Peter
102 #Celine

Multi-Graph Queries and Joins

```
CONSTRUCT (c)<-[:worksAt]-(n)
MATCH (c:Company) ON company_graph,
      (n:Person) ON social_graph
WHERE c.name = n.employer
UNION social_graph
```

c	n
0 #Google	105 #John
1 #HPI	104 #Frank
2 #SAP	102 #Celine
3 #HP	102 #Celine



Multi-Graph Queries and Joins

```
CONSTRUCT (c)<-[:worksAt]-(n)
MATCH (c:Company) ON company_graph,
      (n:Person) ON social_graph
WHERE c.name = n.employer
UNION social_graph
```

c	n
0 #Google	105 #John
1 #HPI	104 #Frank
2 #SAP	102 #Celine
3 #HP	102 #Celine

firstName: Celine
lastName: Mayer
employer: {SAP,HPI}

firstName: Peter
lastName: Smith

firstName: Frank
lastName: Gold
employer: HP

firstName: John
lastName: Doe
employer: Google

102 — 202 — 103 — 203 — 104 — 207 — 204 — 105

0 name:HPI

1 name:SAP

2 name: Google

3 name:HP

Node Labels

Person Manager Company

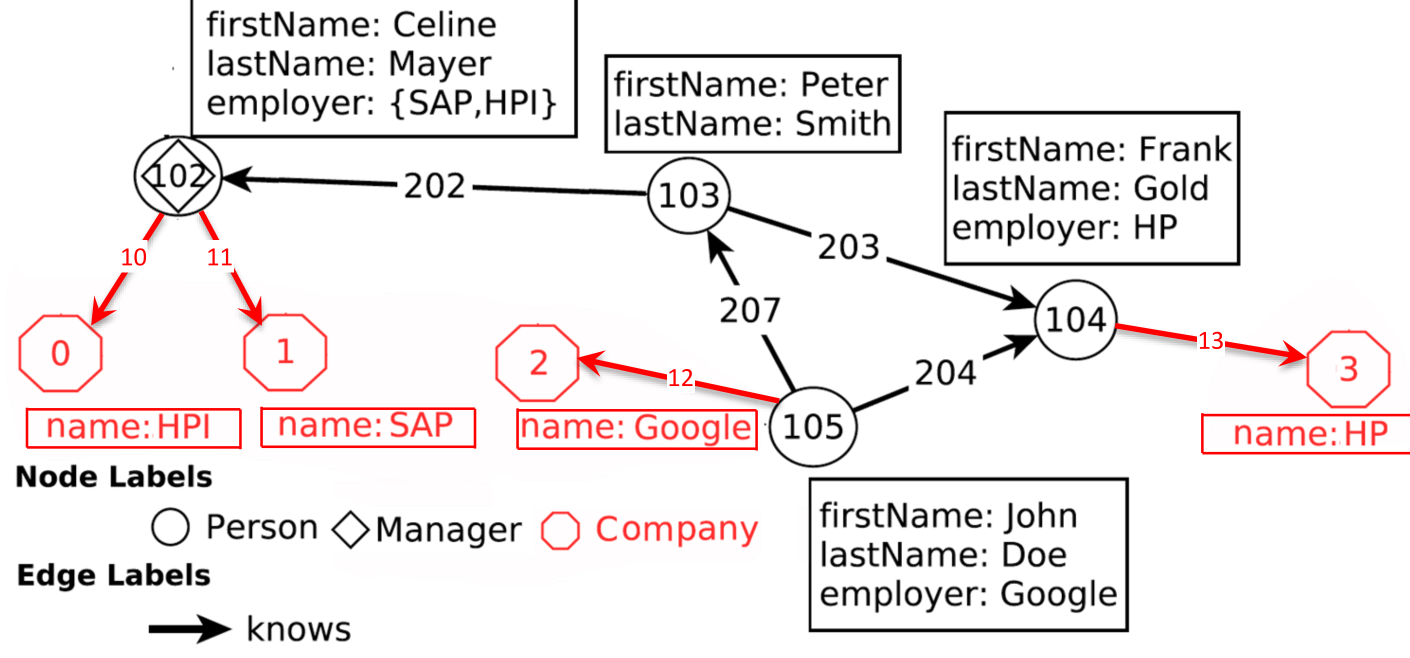
Edge Labels

knows

Multi-Graph Queries and Joins

```
CONSTRUCT (c)<-[:worksAt]-(n)
MATCH (c:Company) ON company_graph,
      (n:Person) ON social_graph
WHERE c.name = n.employer
UNION social_graph
```

c	n
0 #Google	105 #John
1 #HPI	104 #Frank
2 #SAP	102 #Celine
3 #HP	102 #Celine



Graph Construction

- Normalize Data, turn property values into nodes

```
CONSTRUCT social_graph,
 (n)-[y:worksAt]->(x:Company {name:=n.employer})
MATCH (n:Person) ON social_graph
```

- The **unbound** destination node **x** would create a company node for each match result (tuple in binding table).
- This is not what we want: we want only one company per unique name ... So ...

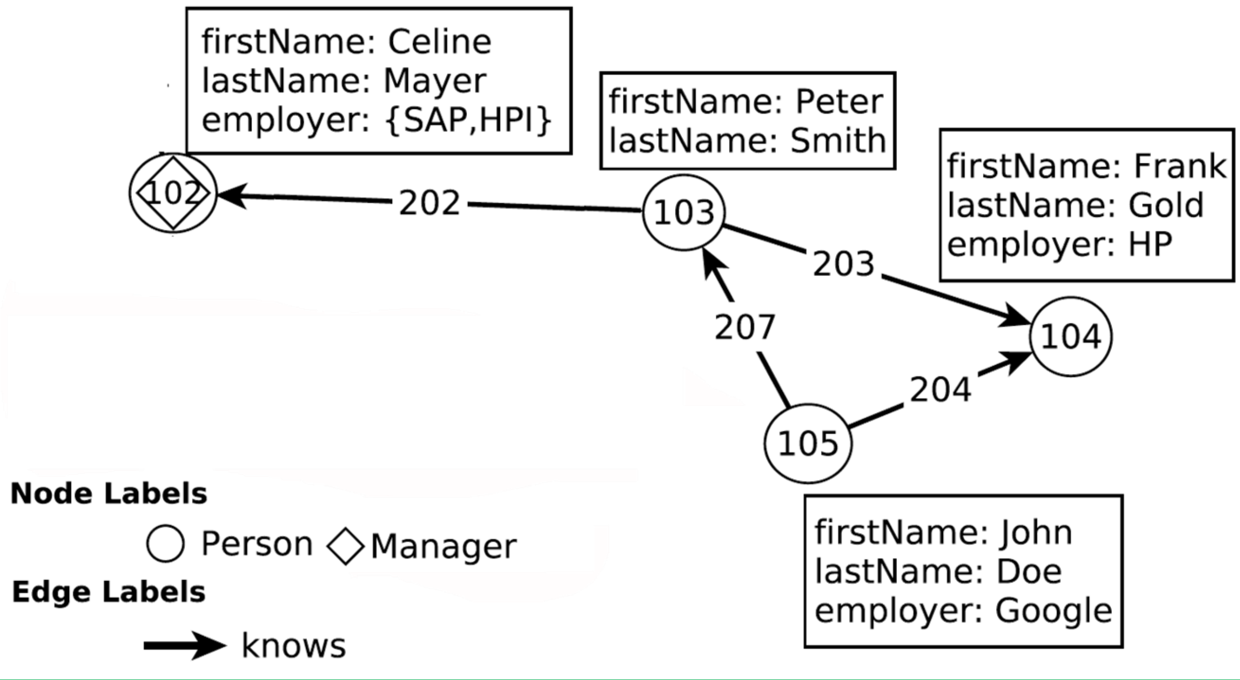
Graph Construction = Graph Aggregation

```
CONSTRUCT social_graph,
 (n)-[y:worksAt]->(x GROUP e :Company {name=e})
MATCH (n:Person {employer=e}) ON social_graph
```

- Graph aggregation: **GROUP** clause in each graph pattern element
- Result: One company node for each unique value of **e** in the binding set is created

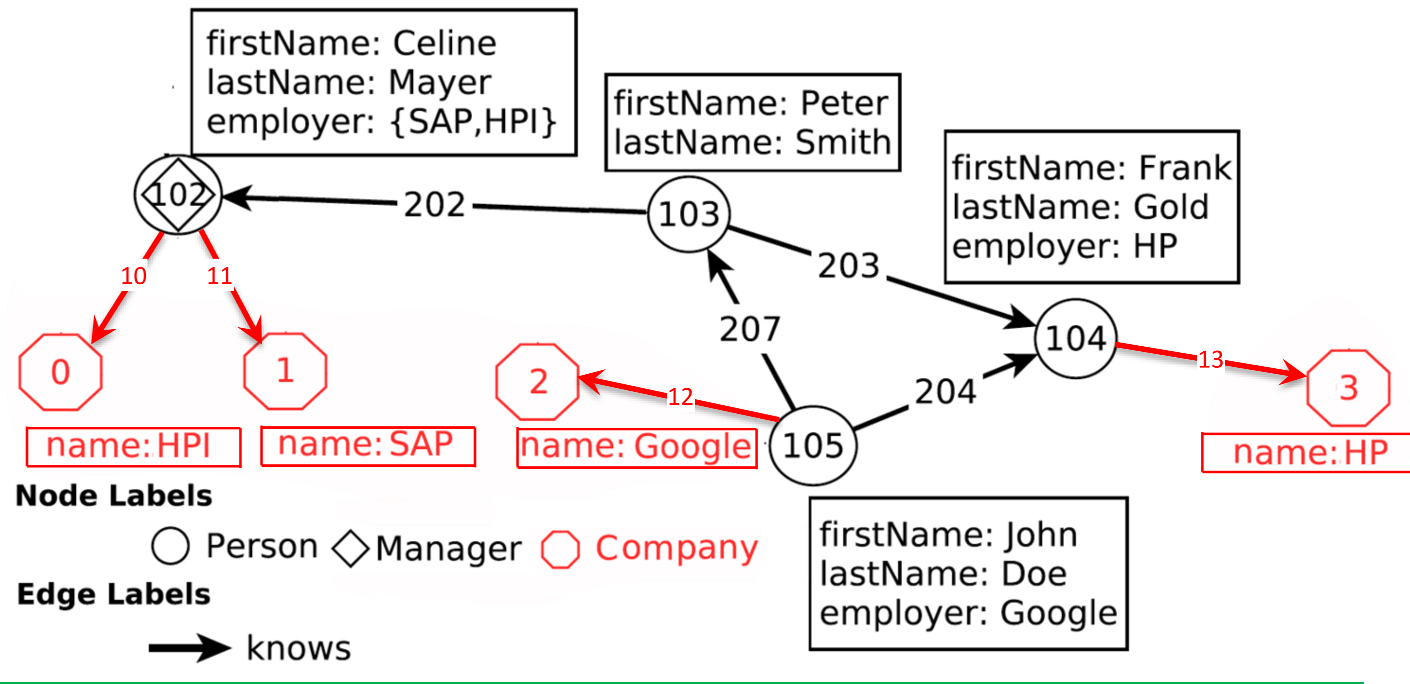
Creating Graphs from Values

```
CONSTRUCT social_graph,
 (n)-[y:worksAt]->(x GROUP e :Company {name=e})
MATCH (n:Person {employer=e}) ON social_graph
```



Creating Graphs from Values

```
CONSTRUCT social_graph,
 (n)-[y:worksAt]->(x GROUP e :Company {name=e})
MATCH (n:Person {employer=e}) ON social_graph
```



Reachability over Paths

- Paths are demarcated with slashes -/ /-
- Regular path expression are demarcated with < >

```
CONSTRUCT (m)
MATCH (n:Person)-/<:knows*>/->(m:Person)
WHERE n.firstName = 'John' AND n.lastName = 'Doe'
  AND (n)-[:isLocatedIn]->()<-[:isLocatedIn]-(m)
```

- If we return just the node (m), the `<:knows*>` path expression semantics is a reachability test

Existential Subqueries

```
CONSTRUCT (m)
MATCH (n:Person)-/<:knows*>/->(m:Person)
WHERE n.firstName = 'John' AND n.lastName = 'Doe'
  AND (n)-[:isLocatedIn]->()<-[:isLocatedIn]-(m)
```

Syntactical shorthand for existential subquery:

```
WHERE …
      EXISTS (
        CONSTRUCT ()
        MATCH (n)-[:isLocatedIn]->()<-[:isLocatedIn]-(m)
      )
```

Storing Paths with @p

- Save the three shortest paths from John Doe towards other person who lives at his location, reachable over knows edges

```
CONSTRUCT (n)-/@p:localPeople{distance:=c}/->(m)
MATCH (n)-/3 SHORTEST p <:knows*> COST c/->(m)
WHERE n.firstName = 'John' AND n.lastName = 'Doe'
  AND (n)-[:isLocatedIn]->()<-[:isLocatedIn]-(m)
```

- @ prefix indicates a stored path: query delivers a graph with paths
- paths have *label* :localPeople and cost as *property* ‘distance’
  - Default cost of a path is its hop-count (length)

More G-CORE..

More features: most advanced GQL so far. See SIGMOD 2018 paper!

```
GRAPH VIEW social_graph1 AS (
  CONSTRUCT social_graph, (n)-[e]->(m)
        SET e.nr_messages := COUNT(*)
  MATCH (n)-[e:knows]->(m)
  WHERE (n:Person) AND (m:Person)
  OPTIONAL (n)<-[c1]-(msg1:Post),
           (msg1)-[:reply_of]-(msg2),
           (msg2:Post)-[c2]->(m)
            WHERE (c1:has_creator) AND (c2:has_creator)
)
PATH wKnows = (x)-[e:knows]->(y)
     WHERE NOT 'Google' IN y.employer
     COST 1 / (1 + e.nr_messages)
CONSTRUCT social_graph1, (n)-/@p:toWagner/->(m)
MATCH (n:Person)-/p <~wKnows*>/->(m:Person) ON social_graph1
```

More G-CORE..

- views

```
GRAPH VIEW social_graph1 AS (
  CONSTRUCT social_graph, (n)-[e]->(m)
          SET e.nr_messages := COUNT(*)
  MATCH (n)-[e:knows]->(m)
  WHERE (n:Person) AND (m:Person)
  OPTIONAL (n)<-[c1]-(msg1:Post),
           (msg1)-[:reply_of]-(msg2),
           (msg2:Post)-[c2]->(m)
            WHERE (c1:has_creator) AND (c2:has_creator)
)
PATH wKnows = (x)-[e:knows]->(y)
     WHERE NOT 'Google' IN y.employer
     COST 1 / (1 + e.nr_messages)
CONSTRUCT social_graph1, (n)-/@p:toWagner/->(m)
MATCH (n:Person)-/p <~wKnows*>/->(m:Person) ON social_graph1
```

More G-CORE..

- set-clause in construct

```
GRAPH VIEW social_graph1 AS (
  CONSTRUCT social_graph, (n)-[e]->(m)
        SET e.nr_messages := COUNT(*)
  MATCH (n)-[e:knows]->(m)
  WHERE (n:Person) AND (m:Person)
  OPTIONAL (n)<-[c1]-(msg1:Post),
           (msg1)-[:reply_of]-(msg2),
           (msg2:Post)-[c2]->(m)
            WHERE (c1:has_creator) AND (c2:has_creator)
)
PATH wKnows = (x)-[e:knows]->(y)
     WHERE NOT 'Google' IN y.employer
     COST 1 / (1 + e.nr_messages)
CONSTRUCT social_graph1, (n)-/@p:toWagner/->(m)
MATCH (n:Person)-/p <~wKnows*>/->(m:Person) ON social_graph1
```

More G-CORE..

- optional match

```
GRAPH VIEW social_graph1 AS (
  CONSTRUCT social_graph, (n)-[e]->(m)
        SET e.nr_messages := COUNT(*)
  MATCH (n)-[e:knows]->(m)
  WHERE (n:Person) AND (m:Person)
  OPTIONAL (n)<-[c1]-(msg1:Post),
           (msg1)-[:reply_of]-(msg2),
           (msg2:Post)-[c2]->(m)
            WHERE (c1:has_creator) AND (c2:has_creator)
)
PATH wKnows = (x)-[e:knows]->(y)
     WHERE NOT 'Google' IN y.employer
     COST 1 / (1 + e.nr_messages)
CONSTRUCT social_graph1, (n)-/@p:toWagner/->(m)
MATCH (n:Person)-/p <~wKnows*>/->(m:Person) ON social_graph1
```

More G-CORE..

- regular path expressions (flexible Kleene*)

```
GRAPH VIEW social_graph1 AS (
  CONSTRUCT social_graph, (n)-[e]->(m)
         SET e.nr_messages := COUNT(*)
  MATCH (n)-[e:knows]->(m)
  WHERE (n:Person) AND (m:Person)
  OPTIONAL (n)<-[c1]-(msg1:Post),
           (msg1)-[:reply_of]-(msg2),
           (msg2:Post)-[c2]->(m)
            WHERE (c1:has_creator) AND (c2:has_creator)
)
PATH wKnows = (x)-[e:knows]->(y)
      WHERE NOT 'Google' IN y.employer
      COST 1 / (1 + e.nr_messages)
CONSTRUCT social_graph1, (n)-/@p:toWagner/->(m)
MATCH (n:Person)-/p <~wKnows*>/->(m:Person) ON social_graph1
```

G-CORE+SQL

- allow **SELECT** clause. You form property expressions (x.prop) on variables (x) from the binding table.
- allow **FROM** clause. Columns are single-value properties on the table variable, rest is NULL.
- allow queries that have both **MATCH** and **FROM.** combine with Cartesian Product, as usual.

Result:

- G-CORE+SQL can query **and return** both tables and graphs

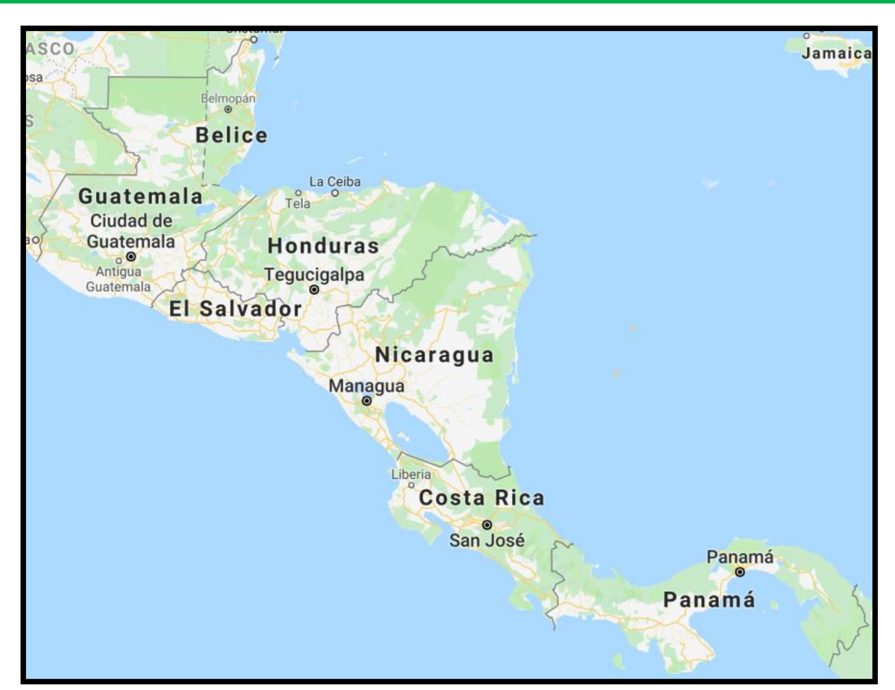
G-CORE Take-Aways

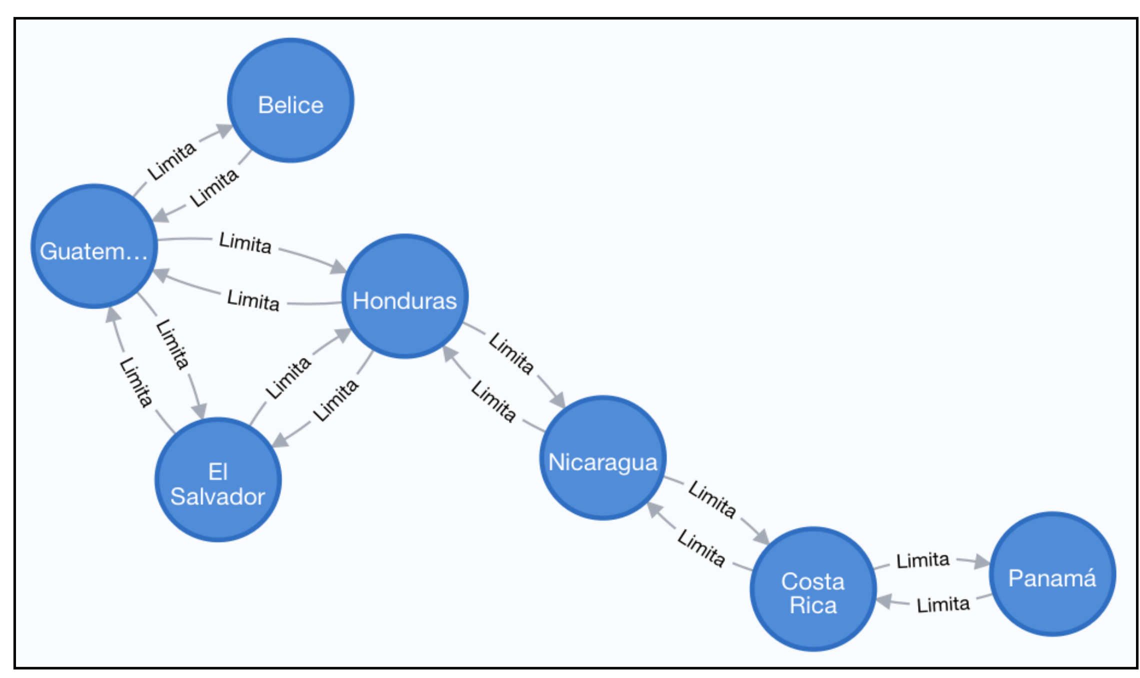
1. G-CORE is a compositional query language for graph data
2. G-CORE can find paths

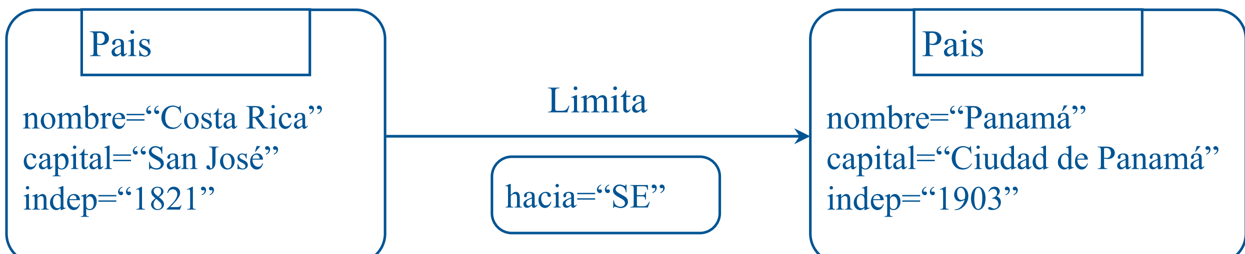
   1+2 = the data model of G-CORE is graphs-with-paths (PPG)

- G-CORE is tractable in data complexity
- G-CORE has many advanced features, e.g.:
  - regular path expressions, views, subqueries ➔ read the paper ☺
- G-CORE+SQL work well together

Comparison of G-CORE, Cypher & Gremlin







G-CORE, Cypher

Cypher Example

```
$ match (n1)-[e]->(n2) return n1
```

$ match (n1)-[e]->(n2) return n1

Graph | Table | Text | Code

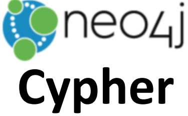
*(7) Pais(7)

*(14) Limita(14)

Nicaragua — Limita — Honduras
Nicaragua — Limita — Costa Rica
Costa Rica — Limita — Panamá
Honduras — Limita — Guatem…
Honduras — Limita — El Salvador
El Salvador — Limita — Guatem…
Guatem… — Limita — Belice

Displaying 7 nodes, 14 relationships.

$ match (n1)-[e]->(n2) return n1



Cypher Example

```
$ match (n1)-[e]->(n2) return n1
```

```
$ match (n1)-[e]->(n2) return n1
```

Graph | Table | Text | Code

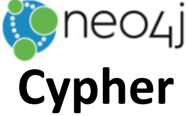
n1

```
{
  "indep": "1821",
  "nombre": "Guatemala",
  "capital": "Ciudad de Guatemala"
}
```

```
{
  "indep": "1821",
  "nombre": "Guatemala",
  "capital": "Ciudad de Guatemala"
}
```

```
{
  "indep": "1821",
```

Started streaming 14 records after 1 ms and completed after 2 ms.



Cypher

Cypher Example

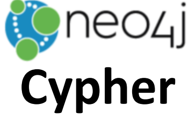
```
$ match (n1)-[e]->(n2) return n1
```

$ match (n1)-[e]->(n2) return n1

Graph
Table
Text
Code

"n1"
{"indep":"1821","nombre":"Guatemala","capital":"Ciudad de Guatemala"}
{"indep":"1821","nombre":"Guatemala","capital":"Ciudad de Guatemala"}
{"indep":"1821","nombre":"Guatemala","capital":"Ciudad de Guatemala"}
{"indep":"1981","nombre":"Belice","capital":"Belmopan"}
{"indep":"1821","nombre":"Honduras","capital":"Tegucigalpa"}
{"indep":"1821","nombre":"Honduras","capital":"Tegucigalpa"}
{"indep":"1821","nombre":"Honduras","capital":"Tegucigalpa"}
{"indep":"1821","nombre":"El Salvador","capital":"San Salvador"}
{"indep":"1821","nombre":"El Salvador","capital":"San Salvador"}
{"indep":"1821","nombre":"Nicaragua","capital":"Managua"}
{"indep":"1821","nombre":"Nicaragua","capital":"Managua"}
{"indep":"1821","nombre":"Costa Rica","capital":"San José"}

MAX COLUMN WIDTH:



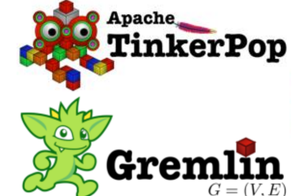
Cypher

Gremlin Examples

```
gremlin> g.V()
==>v[1]
==>v[2]
==>v[3]
==>v[4]
==>v[5]
==>v[6]
==>v[7]
```

```
gremlin> g.E()
==>e[32][6-Limita->5]
==>e[33][6-Limita->7]
==>e[34][7-Limita->6]
==>e[21][1-Limita->2]
==>e[22][1-Limita->4]
==>e[23][1-Limita->3]
==>e[24][2-Limita->1]
==>e[25][3-Limita->1]
==>e[26][3-Limita->4]
==>e[27][3-Limita->5]
==>e[28][4-Limita->1]
==>e[29][4-Limita->3]
==>e[30][5-Limita->3]
==>e[31][5-Limita->6]
```

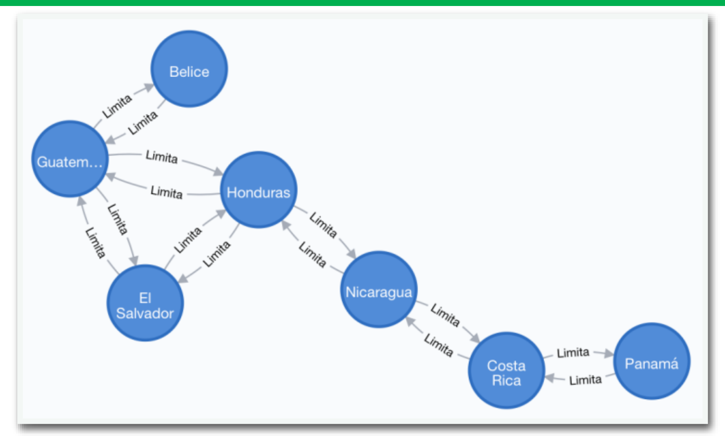
```
gremlin> g.V().has('nombre','Nicaragua').outE('Limita').inV().values('nombre')
==>Honduras
==>Costa Rica
```

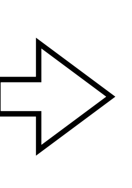


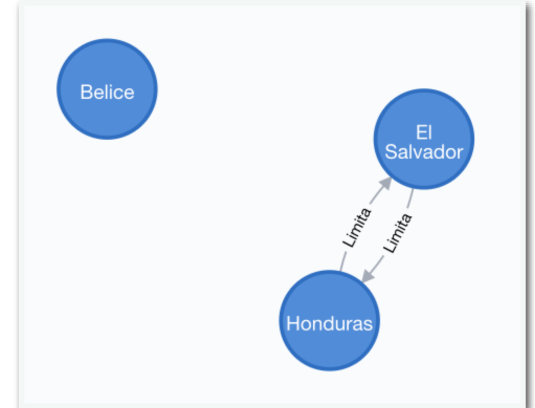
Adjacency

Query:

Return the neighboring countries of Guatemala







cypher

```
MATCH (x)-[e:Limita]->(y)
WHERE x.nombre='Guatemala'
RETURN y
```

gremlin

```
g.V().has('nombre','Guatemala').out('Limita')
```

g-core

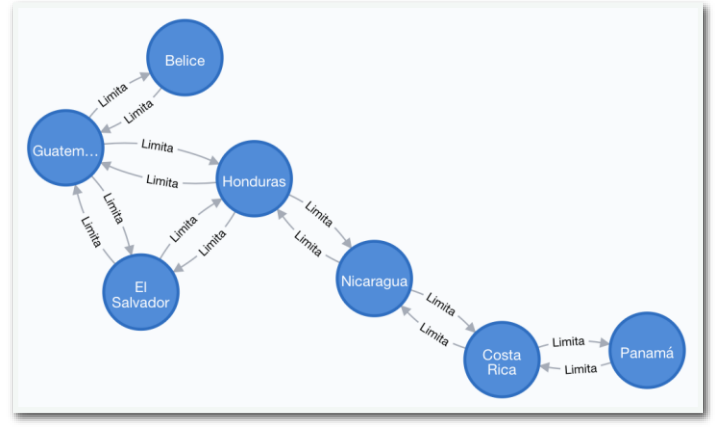
```
CONSTRUCT (y)
MATCH (x)-[]->(y) ON mygraph
WHERE x.nombre='Guatemala'
```

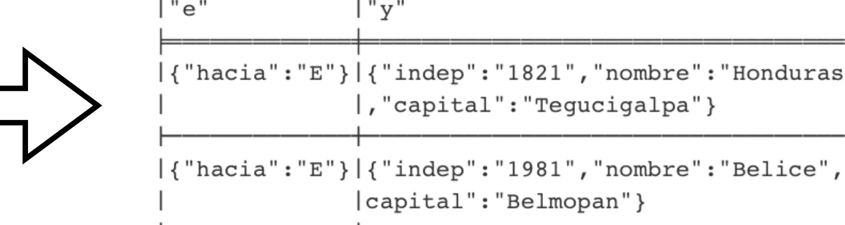
G-CORE, Cypher

Adjacency + Filter

Query:

Return the neighboring countries to the east of Guatemala





"e"	"y"
{"hacia":"E"}	{"indep":"1821","nombre":"Honduras","capital":"Tegucigalpa"}
{"hacia":"E"}	{"indep":"1981","nombre":"Belice","capital":"Belmopan"}

cypher

```
MATCH (x)-[e:Limita]->(y)
WHERE x.nombre = 'Guatemala' AND e.hacia = 'E'
RETURN e, y
```

gremlin

```
g.V().has(‘nombre’,'Guatemala').outE('Limita').has('hacia','E').inV()
```

g-core

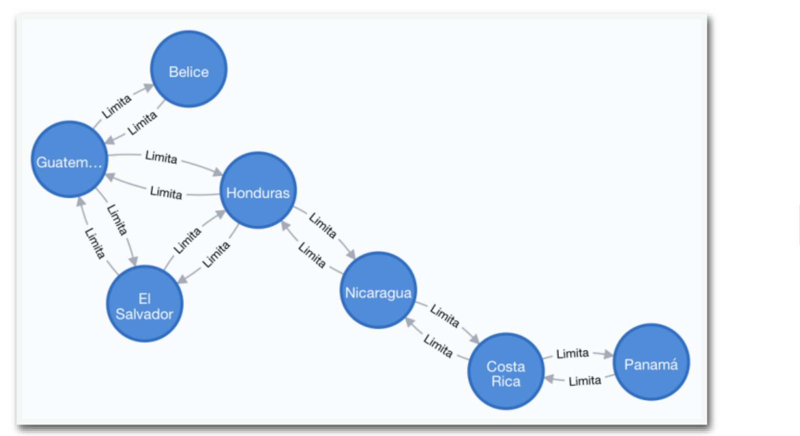
```
CONSTRUCT (y)
MATCH (x)-[e:Limita]->(y)
WHERE x.nombre = ‘Guatemala' AND e.hacia = 'E'
```

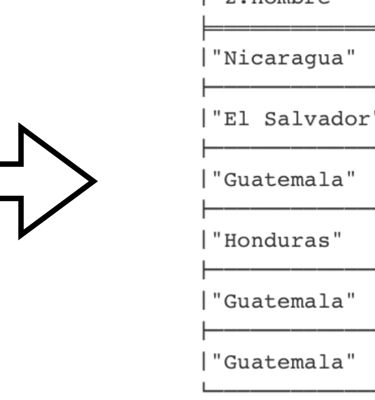
G-CORE, Cypher

Adjacency - Hop

Query:

Return the neighbors of Guatemala at distance 2 (fixed path)





"z.nombre"
"Nicaragua"
"El Salvador"
"Guatemala"
"Honduras"
"Guatemala"
"Guatemala"

cypher

```
MATCH (x)-[e1]->(y)-[e2]->(z)
WHERE x.nombre='Guatemala'
RETURN z.nombre
```

gremlin

```
g.V().has('nombre','Guatemala').out().out().values('nombre')
```

g-core

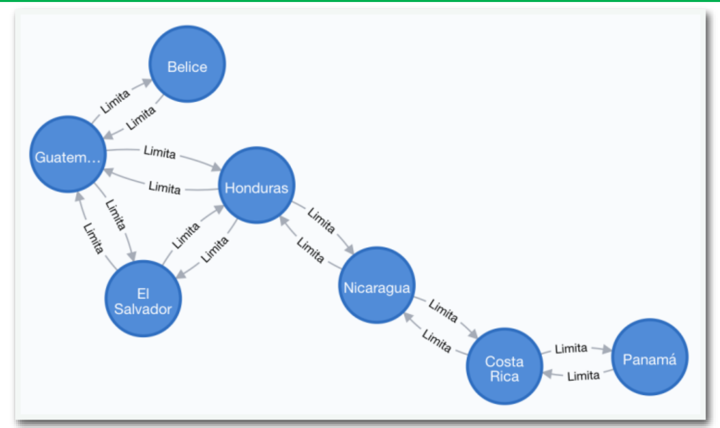
```
CONSTRUCT (z)
MATCH (x)-[e1]->(y)-[e2]->(z)
WHERE x.nombre = 'Guatemala'
```

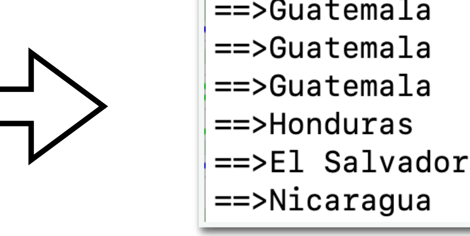
G-CORE, Cypher

Adjacency - Hop

Query:

Return the neighbors of Guatemala at distance 2 (with recursion)





cypher

```
MATCH (x)-[*2..2]->(y)
WHERE x.nombre='Guatemala'
RETURN y.nombre
```

gremlin

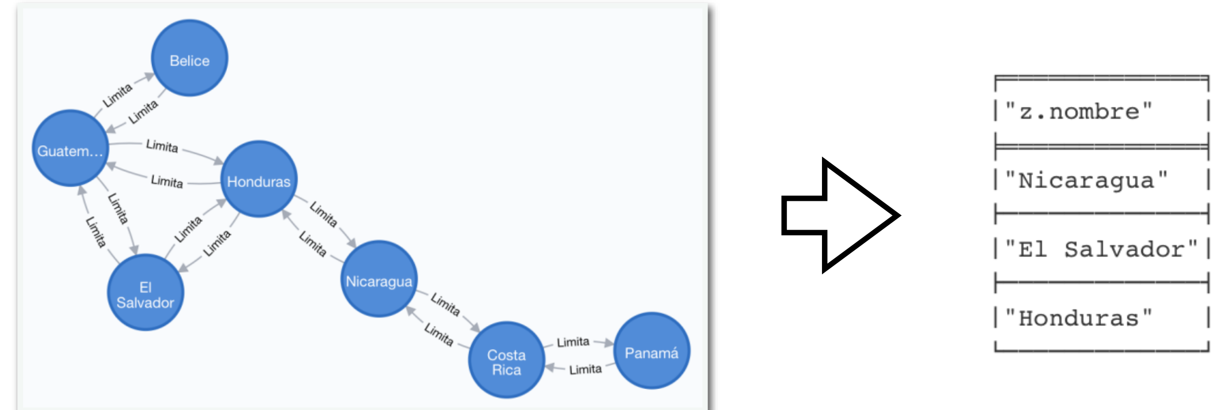
```
g.V().has('nombre','Guatemala').repeat(out()).times(2).values('nombre')
```

g-core

Not supported

G-CORE, Cypher

Adjacency without Duplicates



"z.nombre"
"Nicaragua"
"El Salvador"
"Honduras"

Query:

Return the neighbors of Guatemala at distance 2 without duplicates

cypher

```
MATCH (x)-[e1]->(y)-[e2]->(z)
WHERE x.nombre='Guatemala' AND x<>z
RETURN DISTINCT z.nombre
```

gremlin

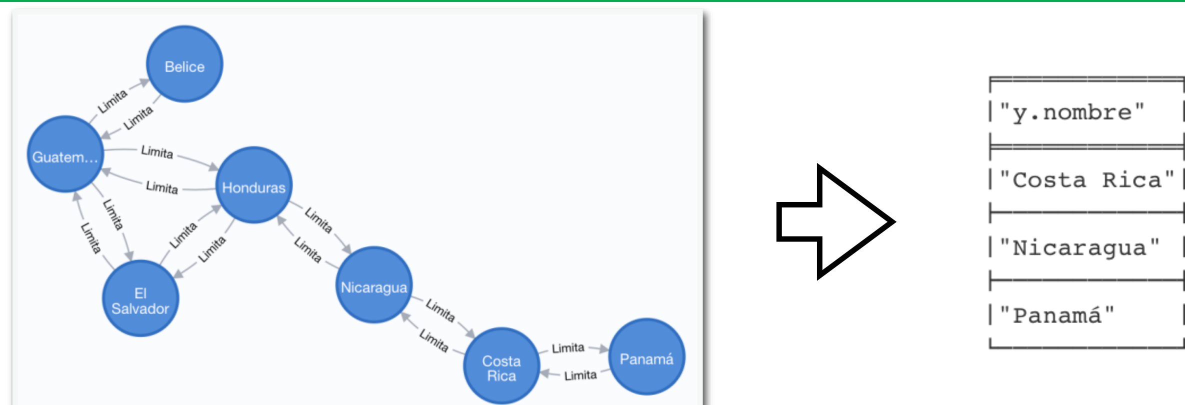
```
g.V().has('nombre','Guatemala').as('no').out().out().where(neq('no'))
.values('nombre')
```

g-core

```
Not supported
```

G-CORE, Cypher

Neighborhood



"y.nombre"
"Costa Rica"
"Nicaragua"
"Panamá"

Query:

Return the neighbors of Guatemala until distance 2

cypher

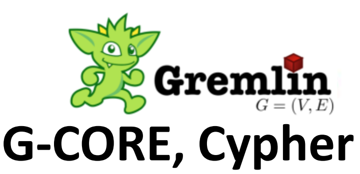
```
MATCH (x)-[*1..2]->(y)
WHERE x.nombre='Panamá'
RETURN y.nombre
```

gremlin

```
g.V().has('nombre','Panamá').repeat(out()).times(2).emit().values('nombre')
```

g-core

```
CONSTRUCT (z) MATCH (x)-[e]->(z) WHERE x.nombre='Panamá'
UNION
CONSTRUCT (z) MATCH (x)-[e1]->(y)-[e2]->(z) WHERE x.nombre='Panamá'
```

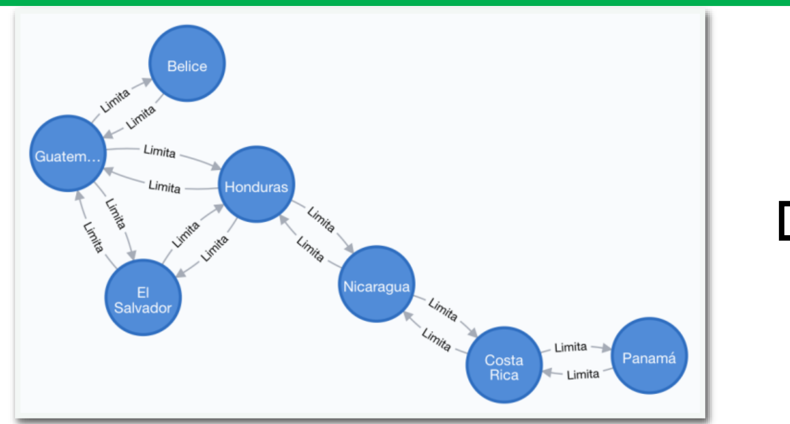


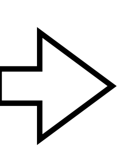
G-CORE, Cypher

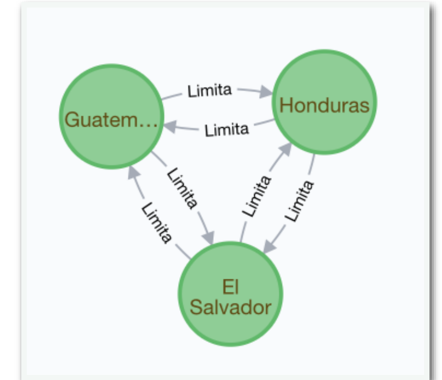
Graph Pattern Matching

Query:

Return three countries all neighboring to each other (triangle query)







cypher

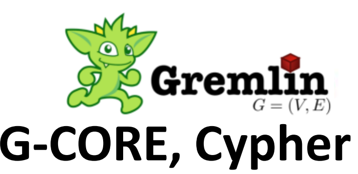
```
MATCH (x)-[:Limita]->(y),(x)-[:Limita]->(z),(y)-[:Limita]->(z)
RETURN x, y, z
```

gremlin

```
g.V().match( __.as('x').out().as('y'), __.as('x').out().as('z'),
__.as('y').out().as('z'),).select('x','y','z').by('nombre')
```

g-core

```
CONSTRUCT (x),(y),(z)
MATCH (x)-[:Limita]->(y),(x)-[:Limita]->(z),(y)-[:Limita]->(z)
```

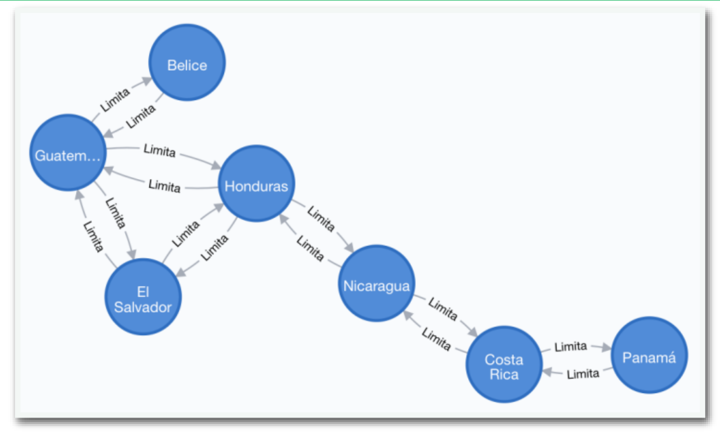


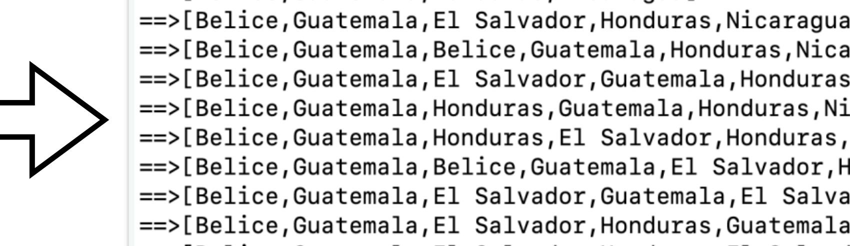
G-CORE, Cypher

Path Finding

Query:

Return all paths between Belice and Nicaragua




```
==>[Belice,Guatemala,Honduras,Nicaragua]
==>[Belice,Guatemala,El Salvador,Honduras,Nicaragua]
==>[Belice,Guatemala,Belice,Guatemala,Honduras,Nicar
==>[Belice,Guatemala,El Salvador,Guatemala,Honduras,
==>[Belice,Guatemala,Honduras,Guatemala,Honduras,Nic
==>[Belice,Guatemala,Honduras,El Salvador,Honduras,N
==>[Belice,Guatemala,Belice,Guatemala,El Salvador,Ho
==>[Belice,Guatemala,El Salvador,Guatemala,El Salvad
==>[Belice,Guatemala,El Salvador,Honduras,Guatemala,
```

cypher

```
MATCH p = (x)-[*]->(y)
WHERE x.nombre='Belice' AND y.nombre='Nicaragua'
RETURN p
```

gremlin

```
g.V().has('nombre','Belice').repeat(timeLimit(5).out()).until(has('no
mbre','Nicaragua')).path().by('nombre')
```

g-core

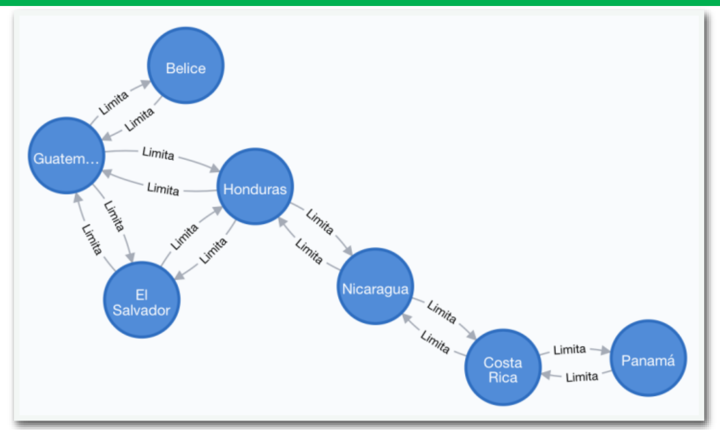
```
RETURN (x)-/p/->(y)
MATCH (x)-/ALL p<:Limita*>/->(y)
WHERE x.nombre = 'Belice' AND y.nombre = 'Nicaragua'
```

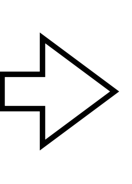
G-CORE, Cypher

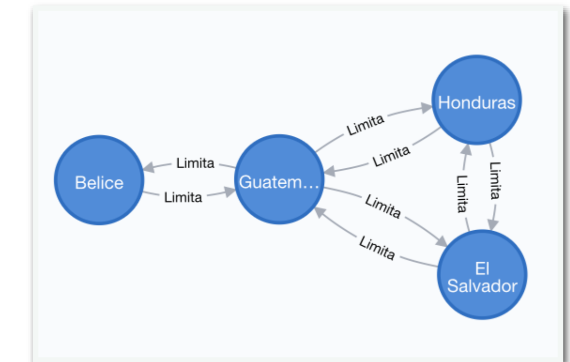
Path Finding (single source)

Query:

Return all path with distance 2 from Belize







cypher

```
MATCH (x {nombre:'Belice'}), p = (x)-[*2..2]->(y)
RETURN p
```

gremlin

```
g.V().has('nombre','Belice').repeat(out()).times(2).path().by('nombre')
```

g-core

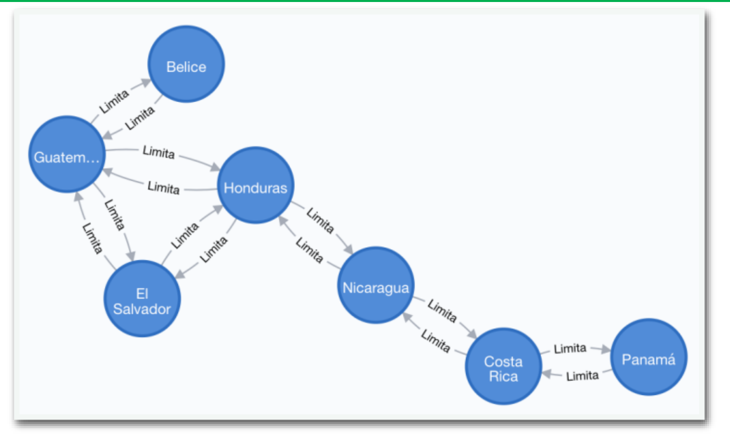
```
RETURN (x)-/p/->(y)
MATCH (x)-/p<:(Limita Limita)>/->(y)
WHERE x.nombre = 'Belice'
```

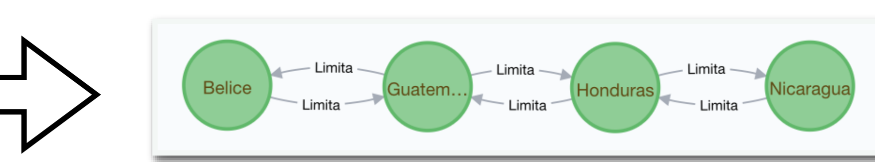
G-CORE, Cypher

Shortest Path

Query:

Return the shortest path between Belice and Panama





cypher

```
MATCH (x {nombre:'Belice'}), (y {nombre:'Nicaragua'}),
      p = shortestPath( (x)-[*]->(y))
RETURN p
```

gremlin

```
g.V().has('nombre','Belice').repeat(out().simplePath()).until(has('nombre','Panamá')).path().by('nombre').limit(1)
```

g-core

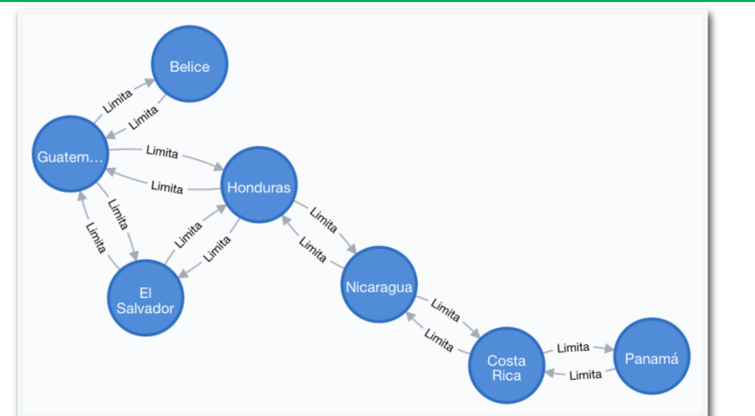
```
RETURN (x)-/p/->(y)
MATCH (x)-/SHORTEST p<:Limita*>/->(y)
WHERE x.nombre = 'Belice' AND y.nombre = 'Nicaragua'
```

G-CORE, Cypher

Global Aggregation

Query:

Return the number of neighbouring countries for Honduras



"count(y)"
3

cypher

```
MATCH (x {nombre:'Honduras'})-[]->(y)
RETURN COUNT(y)
```

gremlin

```
g.V().has('nombre','Honduras').out().count()
```

g-core

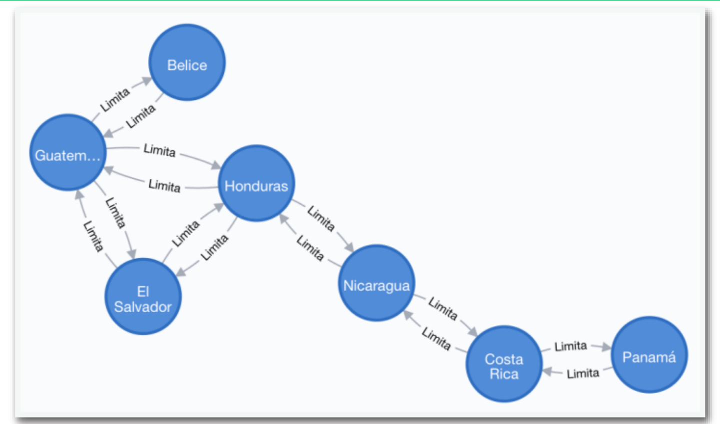
```
CONSTRUCT (x) SET x.number = COUNT(*)
MATCH (x)-[:Limita]->()
WHERE x.nombre = 'Honduras'
```

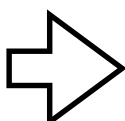
G-CORE, Cypher

Grouping Aggregation

Query:

Return the number of neighboring countries for each country





```
==>[Belice:1,Nicaragua:2,El
Salvador:2,Panamá:1,Guatemala:3
,Honduras:3,Costa Rica:2]
```

cypher

```
MATCH (x)-[]->(y) RETURN x, COUNT(y)
```

gremlin

```
g.V().hasLabel('Pais').group().by('nombre').by(out('Limita').count())
```

g-core

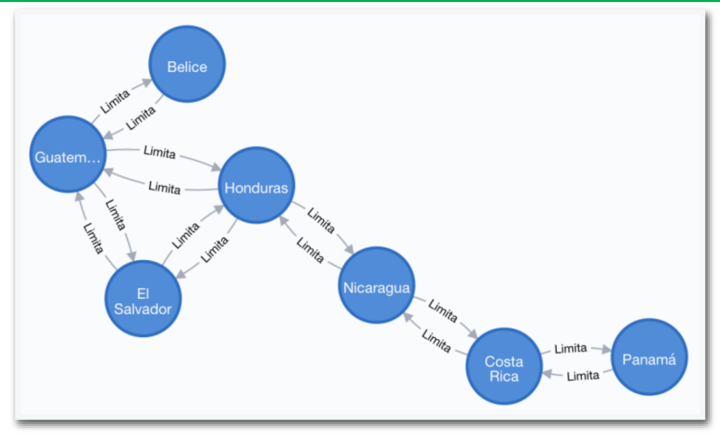
```
CONSTRUCT (x)-[:hasNeighbors]->(v GROUP y :GroupedNeighbors)
          SET v.number = COUNT(*)
MATCH (x)-[:Limita]->(y)
```

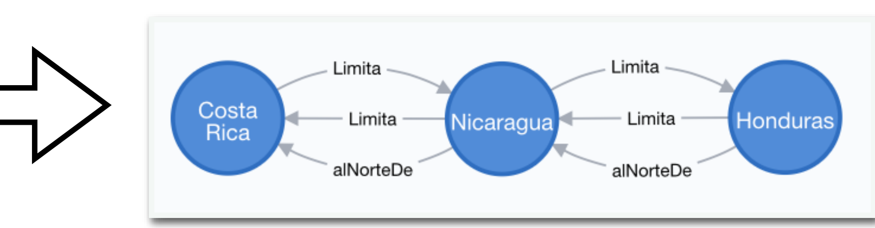
G-CORE, Cypher

Graph Construction

Query:

If Y is a neighbor to the north of X, add an edge labeled 'alNorteDe' between Y and X





cypher

```
MATCH (x)-[e:Limita]->(y)
WHERE e.hacia='N'
MERGE (y)-[:alNorteDe]->(x)
```

gremlin

```
Not supported
```

g-core

```
CONSTRUCT (y)-[:alNorteDe]->(x)
MATCH (x)-[e:Limita]->(y)
WHERE e.hacia = 'N'
```

G-CORE, Cypher

Cypher WITH: concatenate query blocks

Aggregated results have to pass through a `WITH` clause to be able to filter on.

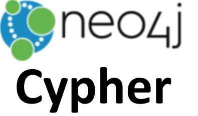
Query.

```
MATCH (david { name: 'David' })--(otherPerson)-->()
WITH otherPerson, count(*) AS foaf
WHERE foaf > 1
RETURN otherPerson.name
```

The name of the person connected to **'David'** with the at least more than one outgoing relationship will be returned by the query.

Table 3.36. Result

otherPerson.name
"Anders"
1 row



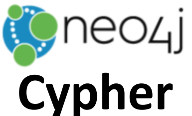
Cypher WITH: concatenate query bocks

```
MATCH (n)
WITH n
ORDER BY n.name DESC LIMIT 3
RETURN collect(n.name)
```

A list of the names of people in reverse order, limited to 3, is returned in a list.

Table 3.37. Result

collect(n.name)
["George","David","Ceasar"]
1 row



Cypher

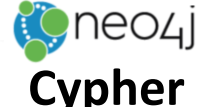
Cypher UNWIND: repeated execution for a list

Multiple `UNWIND` clauses can be chained to unwind nested list elements.

```
WITH [[1, 2],[3, 4], 5] AS nested
UNWIND nested AS x
UNWIND x AS y
RETURN y
```

The first `UNWIND` results in three rows for `x`, each of which contains an element of the original list (two of which are also lists); namely, `[1, 2]`, `[3, 4]` and `5`. The second `UNWIND` then operates on each of these rows in turn, resulting in five rows for `y`.

y
1
2
3
4
5
5 rows



Cypher

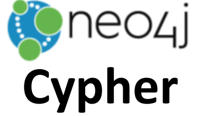
Cypher MERGE: augmenting the graph

For some property 'p' in each bound node in a set of nodes, a single new node is created for each unique value for 'p'.

```
MATCH (person:Person)
MERGE (city:City { name: person.bornIn })
RETURN person.name, person.bornIn, city
```

Three nodes labeled `City` are created, each of which contains a `name` property with the value of **'New York'**, **'Ohio'**, and **'New Jersey'**, respectively. Note that even though the `MATCH` clause results in three bound nodes having the value **'New York'** for the `bornIn` property, only a single **'New York'** node (i.e. a `City` node with a name of **'New York'**) is created. As the **'New York'** node is not matched for the first bound node, it is created. However, the newly-created **'New York'** node is matched and bound for the second and third bound nodes.

person.name	person.bornIn	city
"Charlie Sheen"	"New York"	Node[7]{name:"New York"}
"Martin Sheen"	"Ohio"	Node[8]{name:"Ohio"}
"Michael Douglas"	"New Jersey"	Node[9]{name:"New Jersey"}
"Oliver Stone"	"New York"	Node[7]{name:"New York"}
"Rob Reiner"	"New York"	Node[7]{name:"New York"}
5 rows, Nodes created: 3 Properties set: 3 Labels added: 3		



Cypher

Cypher CALL: subqueries

Variables are imported into a subquery using an importing `WITH` clause. As the subquery is evaluated for each incoming input row, the imported variables get bound to the corresponding values from the input row in each evaluation.

```
UNWIND [0, 1, 2] AS x
CALL {
  WITH x
  RETURN x*10 AS y
}
RETURN x, y
```

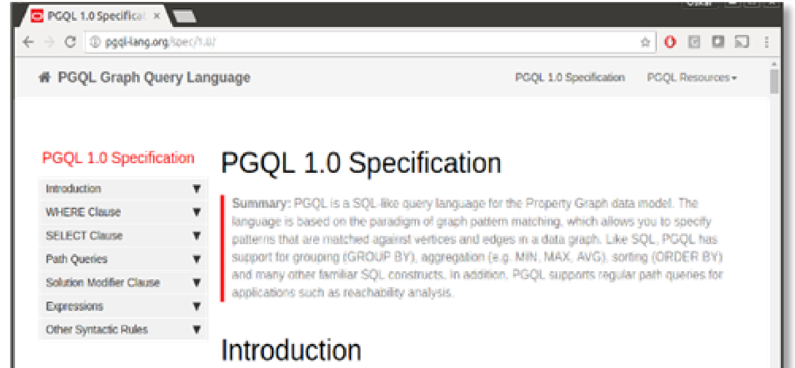
x	y
0	0
1	10
2	20
3 rows	

Cypher

Oracle's PGQL Graph Query Language

- 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

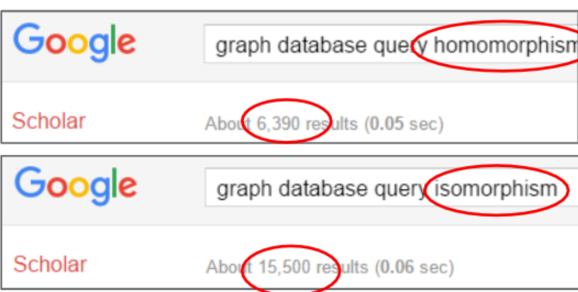


- Implementation (PGQL 1.0)
  - Parallel Graph Analytics (PGX) ORACLE
    - PGX is Oracle's in-memory graph analytics engine
      http://oracle.com/technetwork/oracle-labs/parallel-graph-analytics
    - Component of Oracle Big Data Spatial and Graph
      http://www.oracle.com/database/technologies/bigdata-spatialandgraph.html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

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

Pattern Matching: Homomorphism vs Isomorphism

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

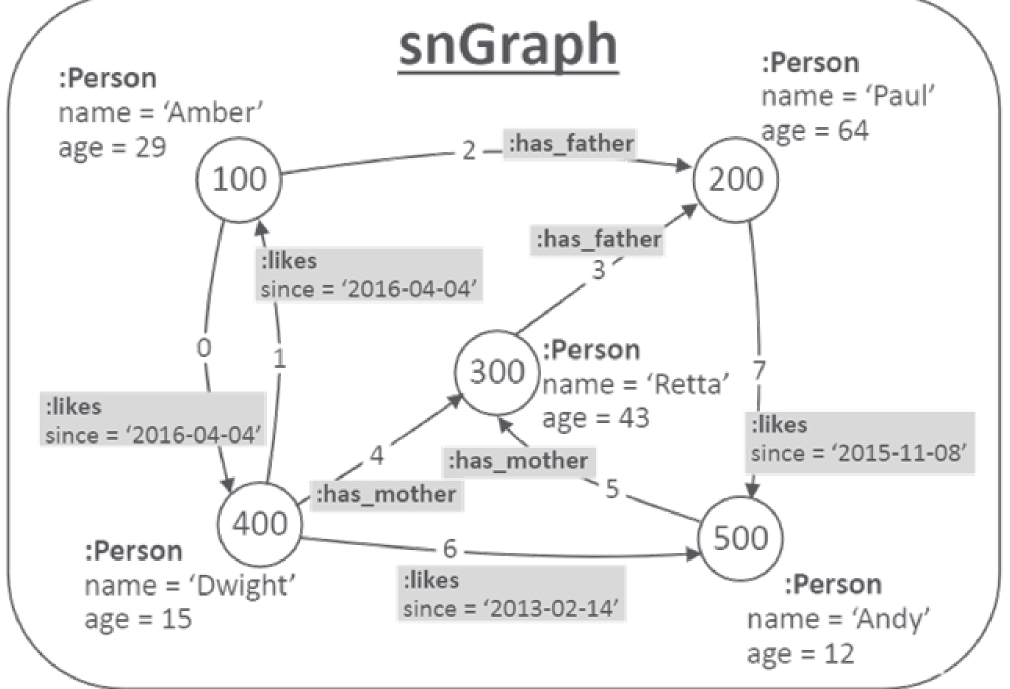


- Isomorphism semantic found to be more intuitive for first-time users
  - *(not based on empirical study)*
  - Homomorphism may return more results than expected (e.g. "find friends of friends of 'John'" returns 'John')
- Isomorphism has limitations (see previous slide)
- Both have the same worst-case 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**

PGQL: Regular Path Query (RPQ) syntax

- 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
```

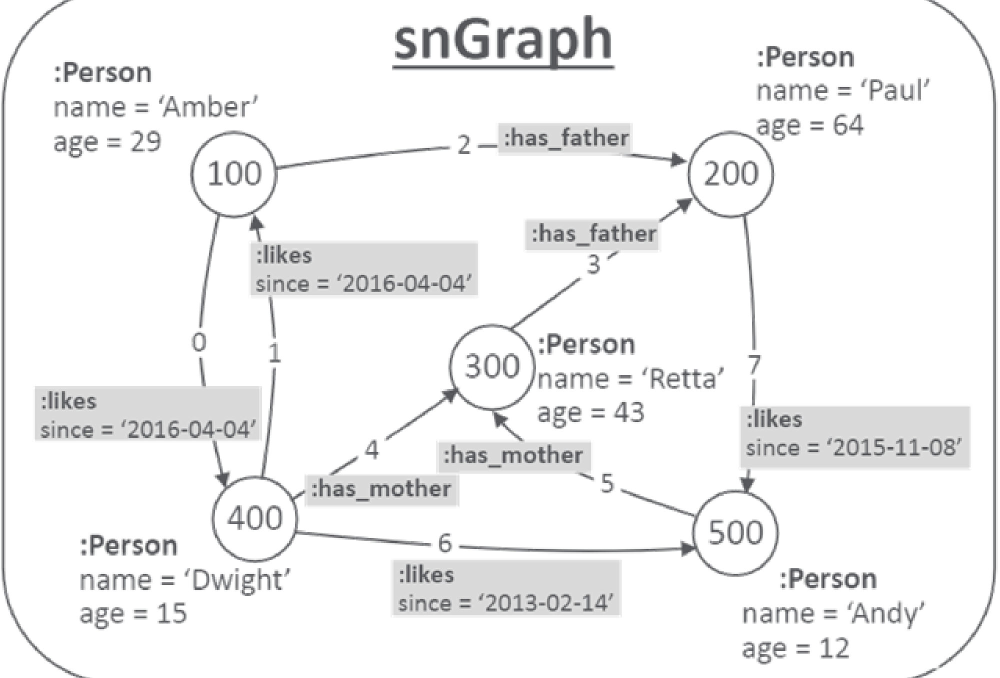


PGQL: Regular Path Query (RPQ) syntax

- 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
```

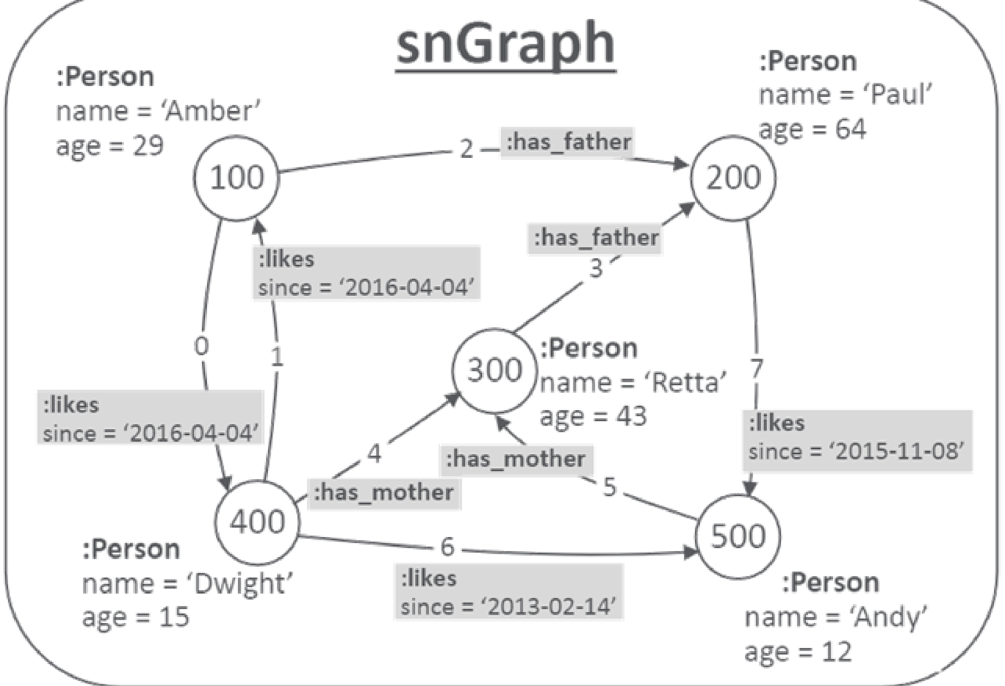
x.id()	y.id()	ancestor.id()
500	300	200
500	400	200
500	400	300



PGQL: Regular Path Query (RPQ) syntax

- 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
```



RPQs: comparing properties along a path

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] https://homepages.inf.ed.ac.uk/libkin/papers/lpar12.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*)

TigerGraph's GSQL: accumulators

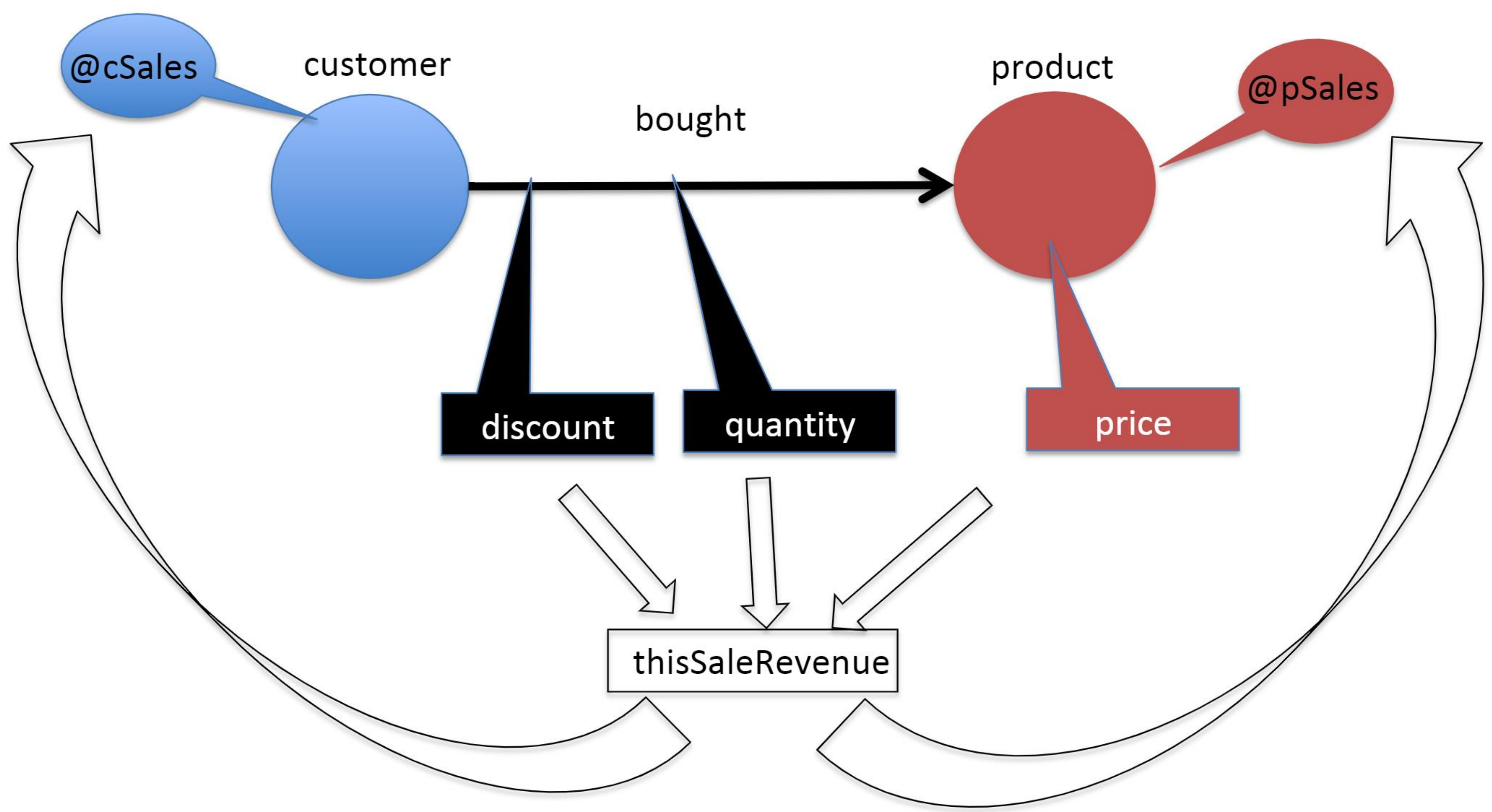
- GSQL traversals collect and aggregate data by writing it into *accumulators*

- Accumulators are containers (data types) that
  - hold a data value
  - accept inputs
  - aggregate inputs into the data value using a binary operator

- May be built-in (sum, max, min, etc.) or user-defined

- May be
  - global (a single container)
  - Vertex-attached (one container per vertex)

GSQL Vertex-attached Aggregators Example



GSQL Vertex-attached Aggregators Example

```
SumAccum<float> @cSales, @pSales;

SELECT   c
FROM     Customer :c −(Bought :b)-> Product :p
ACCUM    thisSaleRevenue = b.quantity*(1-b.discount)*p.price,
         c.@cSales += thisSaleRevenue,
         p.@pSales += thisSaleRevenue;
```

accumulator declaration

groups are distributed, each node accumulates its own group

same sale revenue contributes to two aggregations, each by distinct grouping criteria

Accumulators & Loops: PageRank Example

```
CREATE QUERY pageRank (float maxChange, int maxIteration, float dampingFactor) {

  MaxAccum<float> @@maxDifference = 9999;  // max score change in an iteration
  SumAccum<float> @received_score = 0;      // sum of scores received from neighbors
  SumAccum<float> @score = 1;               // initial score for every vertex is 1.

  AllV = {Page.*};                          // start with all vertices of type Page
  WHILE @@maxDifference > maxChange LIMIT maxIteration DO
   @@maxDifference = 0;

    S= SELECT       s
       FROM         AllV:s -(Linkto)-> :t
       ACCUM        t.@received_score += s.@score/s.outdegree()
       POST-ACCUM   s.@score = 1-dampingFactor + dampingFactor * s.@received_score,
                    s.@received_score = 0,
                    @@maxDifference +=   abs(s.@score - s.@score');
  END;
}
```

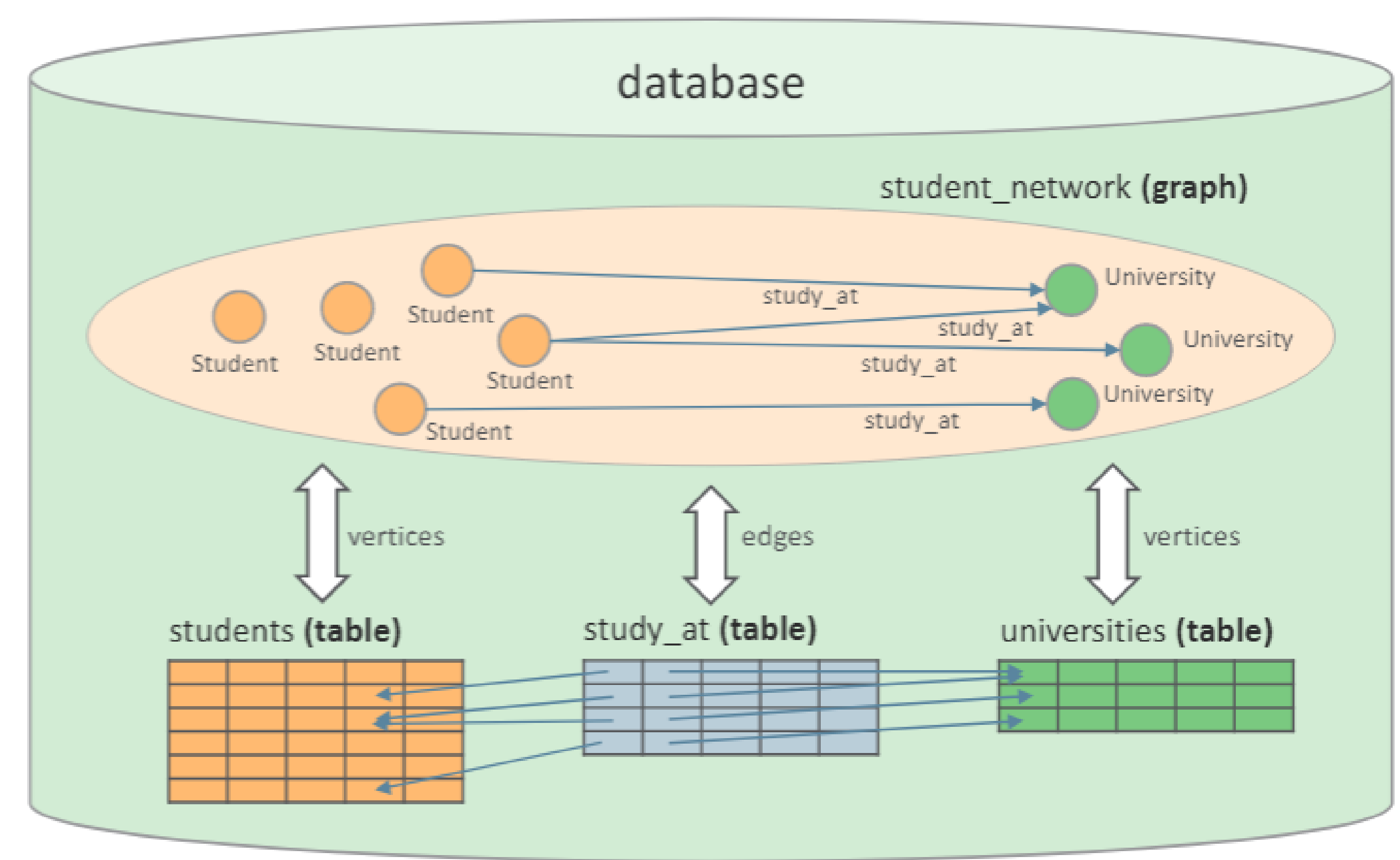
GSQL

SQL/PGQ: SQL Extensions for Property Graphs

- What?
  - Tabular property graph model: store property graphs as sets of tables
  - Graph pattern matching: fixed-length and variable-length (e.g. shortest path)
  - Possibly more, but not in the first version
- Where?
  - Aka the ISO SQL committee
  - ISO: JTC 1/SC32/WG3 (USA, Germany, Japan, UK, Canada, China)
  - Aka the US SQL committee
  - ANSI: INCITS / DM32 / DM32.2 / DM32.2 Ad Hoc Group on SQL Extensions for Property Graphs (Oracle, Neo4j, TigerGraph, IBM, SAP/Sybase, JCC Consulting)
- When?
  - Next version of SQL; possibly SQL:2020 or SQL:2021 (current version is SQL:2016)

Property Graphs That Are Backed By Tables

- A graph is stored as a set of vertex tables and edges tables
- A graph is like a view over existing tables: creating a graph requires no data copying
- There can be multiple graphs per database
- Graphs have a name and live in the same name space as tables



Tables map to sets of vertices and/or edges

- Each row in a vertex/edge table becomes a vertex/edge in the graph
  - By default, table names become labels, but it can be customized
  - By default, all columns become properties, but it can be customized
  - By default, PK-FK relationships are used to create edges, but it can be customized

Example vertex table:

People

id	name	dob
1	Riya	1995-03-20
2	Kathrine	1994-01-15
3	Lee	1996-01-29

myGraph

Person — name: Riya, dob: 1995-03-20

Person — name: Kathrine, dob: 1994-01-15

Person — name: Lee, dob: 1996-01-29

SQL DDL statement:

```
CREATE PROPERTY GRAPH myGraph
  VERTEX TABLES (
    People LABEL Person PROPERTIES ( name, dob )
  )
```

PK-FK relationships in tables to create edges

Vertex tables:

Person

id	name	dob
1	Riya	1995-03-20
2	Kathrine	1994-01-15
3	Lee	1996-01-29

University

id	name
1	UC Berkeley

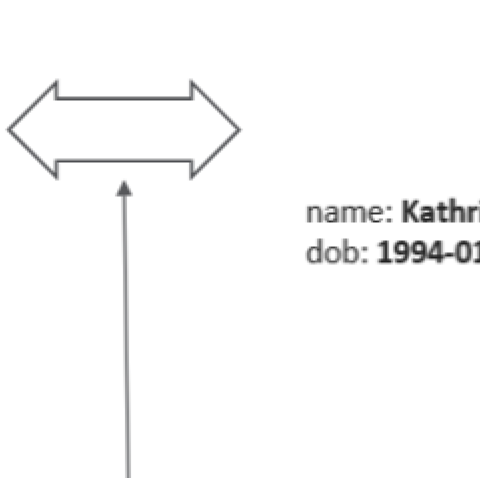
Edge tables:

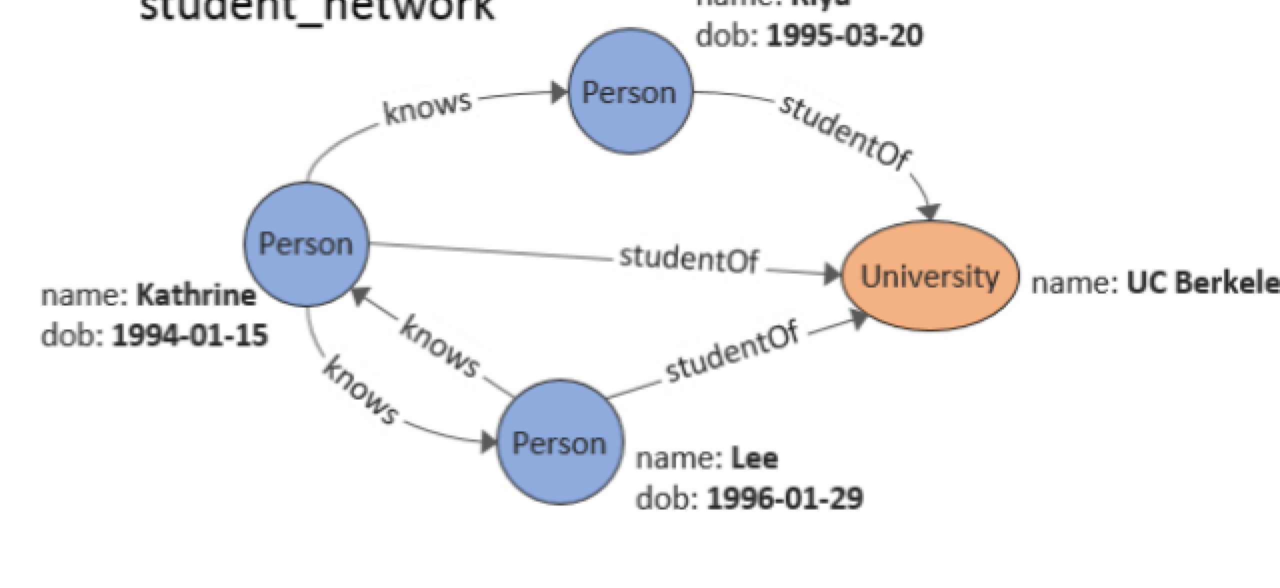
knows

person1_id	person2_id
2	1
2	3
3	2

studentOf

person_id	university_id
1	1
2	1
3	1





SQL DDL statement:

```
CREATE PROPERTY GRAPH student_network
VERTEX TABLES ( Person PROPERTIES ( name, dob ),
                University PROPERTIES ( name ) )
EDGE TABLES ( knows SOURCE Person DESTINATION Person NO PROPERTIES,
              studentOf SOURCE Person DESTINATION University NO PROPERTIES )
```

SQL/PGQ

Manually Specifying keys for vertices/edges

Keys need to be manually specified in case the underlying tables (or views) do not already have the necessary keys defined:

SQL DDL statement:

```
CREATE PROPERTY GRAPH student_network
  VERTEX TABLES ( Person KEY ( id ) PROPERTIES ( name, dob ),
                  University KEY ( id ) PROPERTIES ( name ) )
  EDGE TABLES ( knows SOURCE KEY ( person1_id ) REFERENCES Person
                      DESTINATION KEY ( person2_id ) REFERENCES Person
                      NO PROPERTIES,
                studentOf SOURCE KEY ( person_id ) REFERENCES Person
                          DESTINATION KEY ( university_id ) REFERENCES University
                          NO PROPERTIES )
```

Person

id	name	dob
1	Riya	1995-03-20
2	Kathrine	1994-01-15
3	Lee	1996-01-29

knows

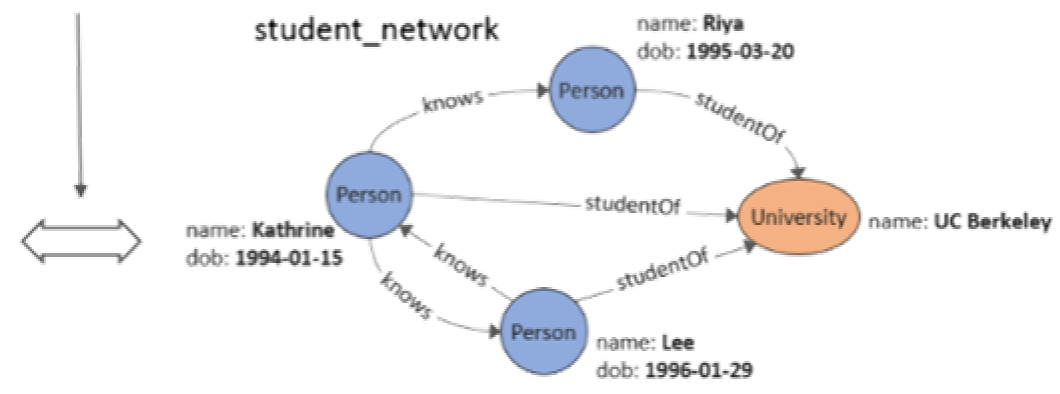
person1_id	person2_id
2	1
2	3
3	2

University

id	name
1	UC Berkeley

studentOf

person_id	university_id
1	1
2	1
3	1



SQL/PGQ

Statically Typed Properties

- Each property belongs to a label:
  - Example with two tables with two labels each:

SQL DDL statement:

```
… VERTEX TABLES ( Students LABEL Person PROPERTIES ( first_name, last_name )
                           LABEL Student PROPERTIES ( student_number ),
                Professors LABEL Person PROPERTIES ( fname AS first_name, last_name )
                           LABEL Professor PROPERTIES ( employee_number ) )
```

Does not rename the underlying column

- Static typing provides safety during querying:

```
MATCH (p IS Personn)
```

Error because no label Personn defined

```
MATCH (p IS Professor)
WHERE p.student_number = ...
```

Error because no property student_number for Professor vertices

```
MATCH (p IS Person)
WHERE p.student_number = ...
```

Will give NULL values for professors but not for students

```
MATCH (p)
WHERE p.student_number = ...
```

Will give NULL values for professors but not for students

Statically Typed Properties

- Each property belongs to a label:
  - Example with two tables with two labels each:

Two vertex/edge tables that share a label need to have the same set of properties defined for that label (same property names and compatible data types)

SQL DDL statement:

```
… VERTEX TABLES ( Students LABEL Person PROPERTIES ( first_name, last_name )
                           LABEL Student PROPERTIES ( student_number ),
                Professors LABEL Person PROPERTIES ( fname AS first_name, last_name )
                           LABEL Professor PROPERTIES ( employee_number ) )
```

Does not rename the underlying column

- Static typing provides safety during querying:

```
MATCH (p IS Personn)
```

Error because no label Personn defined

```
MATCH (p IS Professor)
WHERE p.student_number = ...
```

Error because no property student_number for Professor vertices

```
MATCH (p IS Person)
WHERE p.student_number = ...
```

Will give NULL values for professors but not for students

```
MATCH (p)
WHERE p.student_number = ...
```

Will give NULL values for professors but not for students

SQL/PGQ Example

```
SELECT GT.creationDate, GT.content
FROM myGraph GRAPH_TABLE (
  MATCH
   (Creator IS Person WHERE Creator.email = :email1)
      -[ IS Created ]->
   (M IS Message)
      <-[ IS Commented ]-
   (Commenter IS Person WHERE Commenter.email = :email2)
      WHERE ALL_DIFFERENT (Creator, Commenter)
  COLUMNS (
      M.creationDate,
      M.content )
 ) AS GT
```

SQL/PGQ Example

Get the **creationDate** and **content** of the **messages created by one person** ("email1") and **commented on by another person** ("email2").

```
SELECT GT.creationDate, GT.content
FROM myGraph GRAPH_TABLE (
  MATCH
   (Creator IS Person WHERE Creator.email = :email1)
      -[ IS Created ]->
   (M IS Message)
      <-[ IS Commented ]-
   (Commenter IS Person WHERE Commenter.email = :email2)
      WHERE ALL_DIFFERENT (Creator, Commenter)
  COLUMNS (
      M.creationDate,
      M.content )
 ) AS GT
```

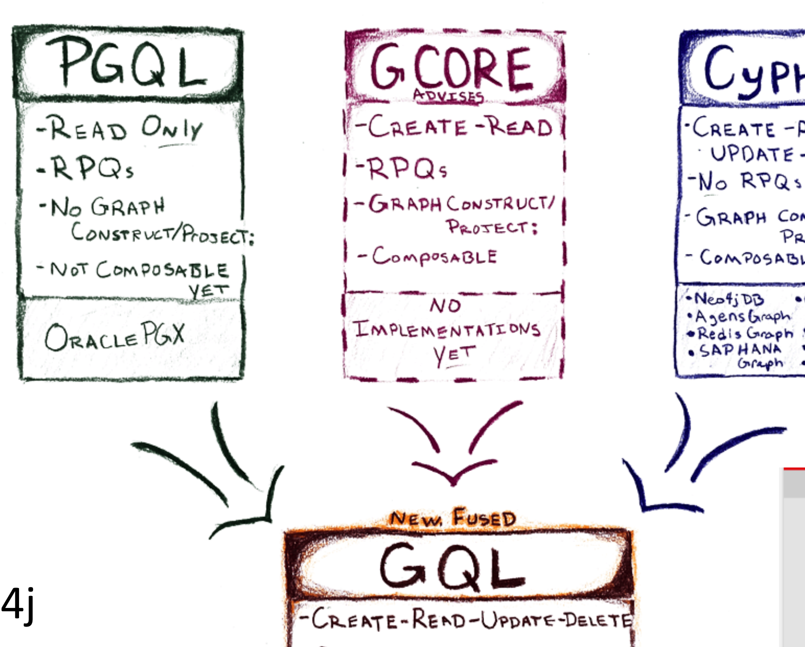
Example: table + graph + CHEAPEST PATH

Given a table with a list of pairs of places called Here and There, for each row in the list, find the cheapest path from Here to There, with a stop at a gas station along the way.

Note: it is possible that some pairs (Here, There) are not connected by a path passing through a gas station; such disconnected pairs must nevertheless be reported in the result. It is possible that Here and There are the same location. It is possible that Here or There or both may be a gas station, in which case it is not necessary to find an additional gas station.

```
SELECT L.Here, GT.GasID, L.There, GT.TotalCost, GT.Eno, GT.Vid GT.Eid
FROM List AS L LEFT OUTER JOIN MyGraph GRAPH_TABLE (
  MATCH CHEAPEST (
      (H IS Place WHERE H.ID = L.Here)
          ( -[R1 IS Route COST R1.Traveltime]-> )*
               (G IS Place WHERE G.HasGas = 1)
           ( -[R2 IS Route COST R2.Traveltime]-> )*
      (T IS Place WHERE T.ID = L.There) )
  ONE ROW PER STEP (V, E)
  COLUMNS ( H.ID AS HID, G.ID AS GasID, T.ID AS TID, TOTAL_COST() AS totalCost,
            ELEMENT_NUMBER (V) AS Eno, V.ID AS Vid, E.ID AS Eid )
) AS GT ON (GT.HID = L.Here AND GT.TID = L.There)
ORDER BY L.Here, L.There, Eno
```

The GQL Manifesto

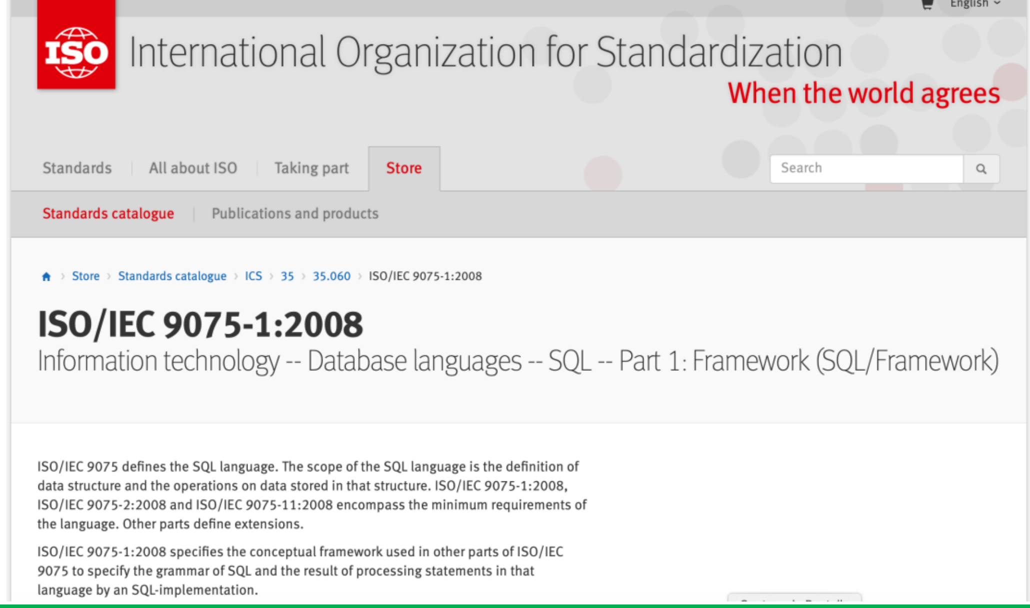


Initiative originally by neo4j

Some coordination in W3C with SPARQL

Main backers (2020) are:
Oracle, neo4j, Microsoft, TigerGraph

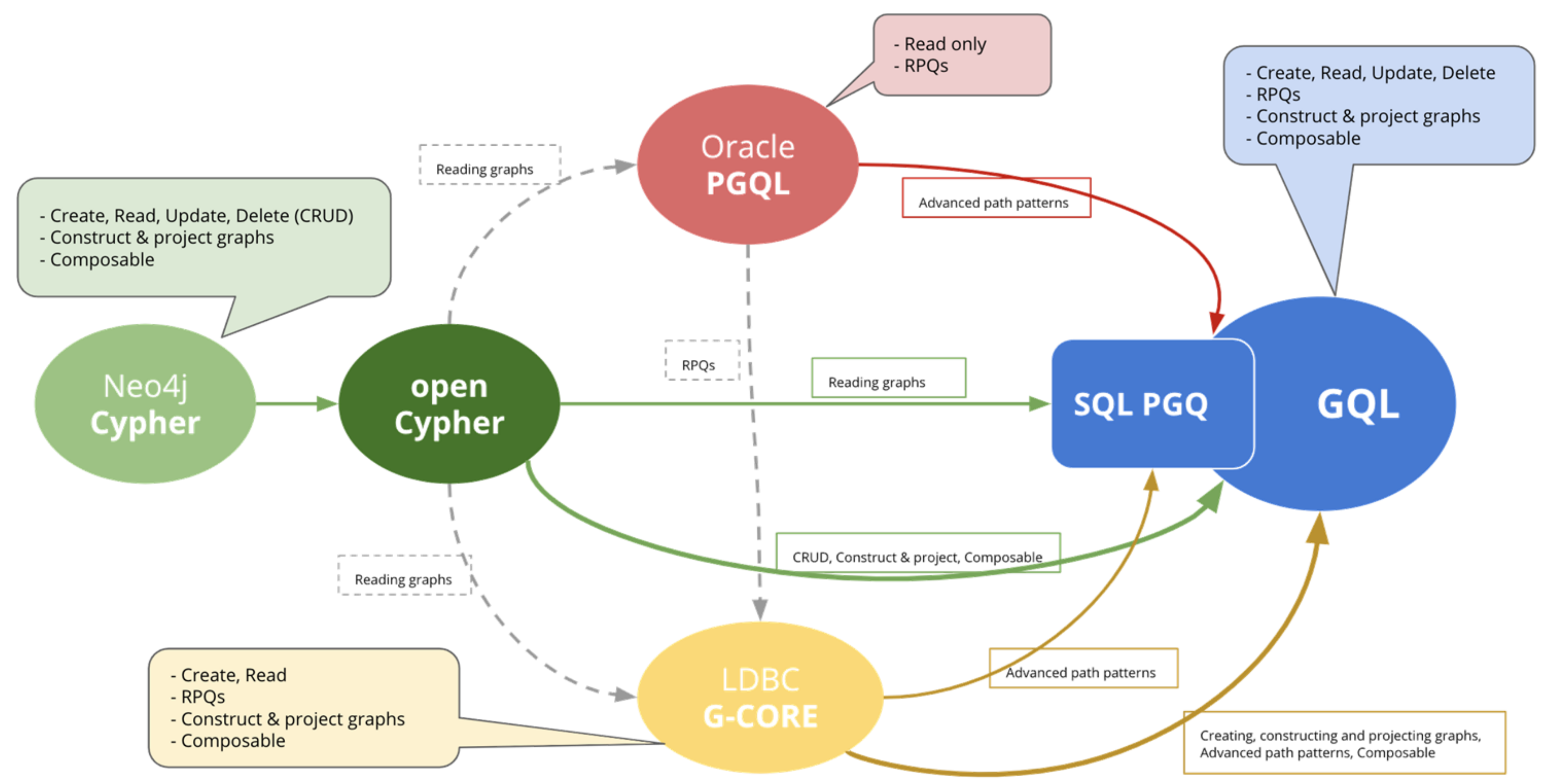
ISO WG led by Jan Michels (Oracle)



ISO: GQL

https://gql.today

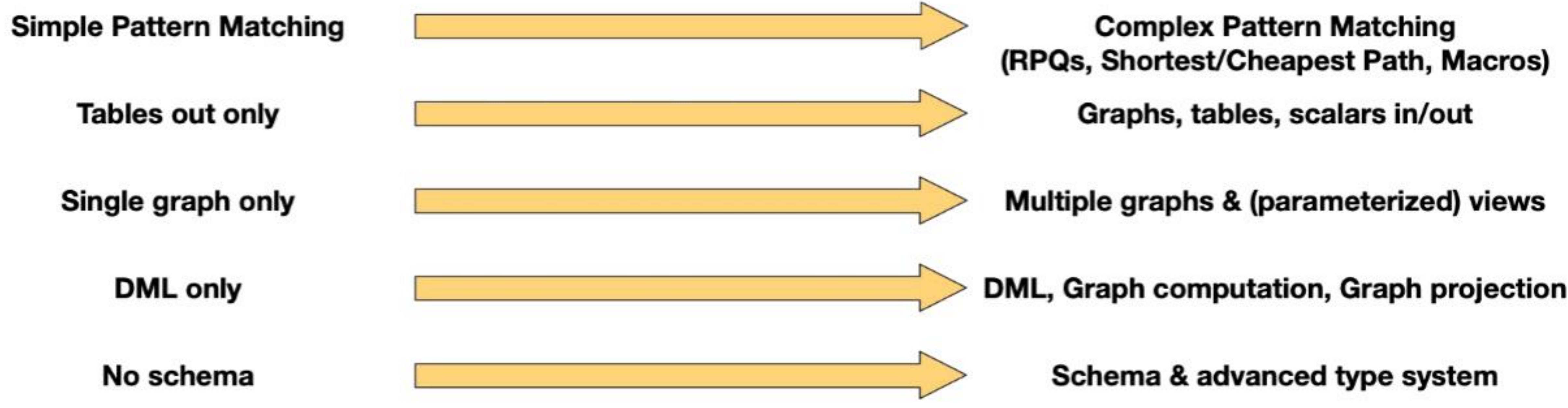
GQL Lineage



ISO: GQL

https://www.gqlstandards.org/home

From Cypher, PGQL, GSQL, SQL/PGQ to **GQL**



All aligned with basic data types, infrastructure, and expressions of the SQL database
Support for basic tabular manipulation (projection, sorting, grouping etc)
More features are discussed (Indexing)

http://tiny.cc/gql-scope-and-features

BNE-023: 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
```

BNE-023: 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)
```

ISO: GQL

BNE-023: 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

```
MATCH ...
```

- `DIFFERENT (VERTICES|NODES)` - vertex isomorphism
- `DIFFERENT (EDGES|RELATIONSHIPS)` - edge isomorphism
- `UNCONSTRAINED` - homomorphism

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

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

ISO: GQL

BNE-023: Why Tabular Operations in GQL?

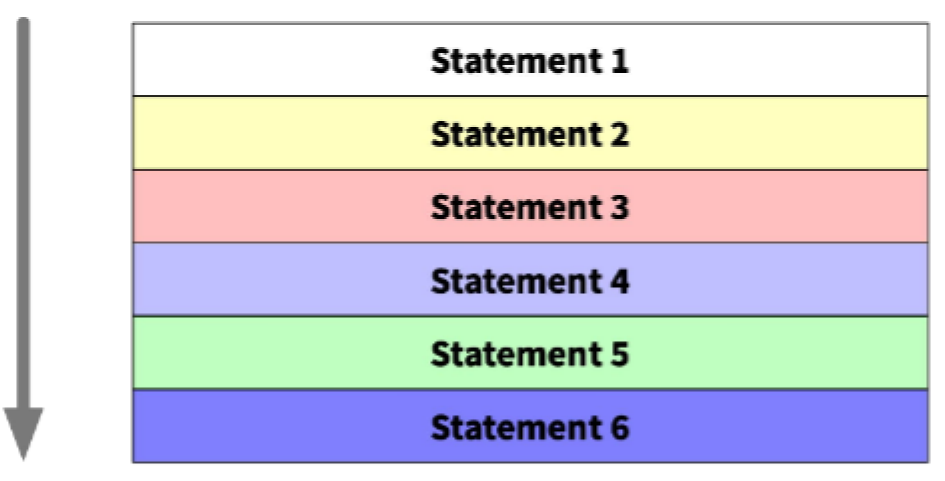
(A) Pattern matching => (Multi) set of bindings (=> Table)
=> *Tabular result transformation useful to avoid client-side processing*

(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

BNE-023: 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)
- **RETURN** has been very positively received by PGM users

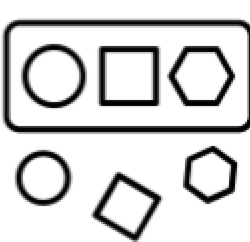
BNE-023: Graph Element Expressions and Functions

- 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

BNE-023: Collection and Dictionary Expressions

- Collection literals: `[ a, b, c, ... ]`
- Dictionary literals: `{ alpha: some(a), beta: b+c, ... }`
- Indexing and lookup: `coll[1], dict['alpha']`
- Map comprehensions
- List comprehension
- Functions

BNE-023:Type System and Schema



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

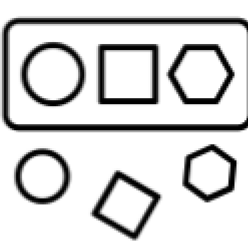
④ Advanced type system features [3.3, 4.4.4]

⑤ Static and dynamic typing [4.4.5]

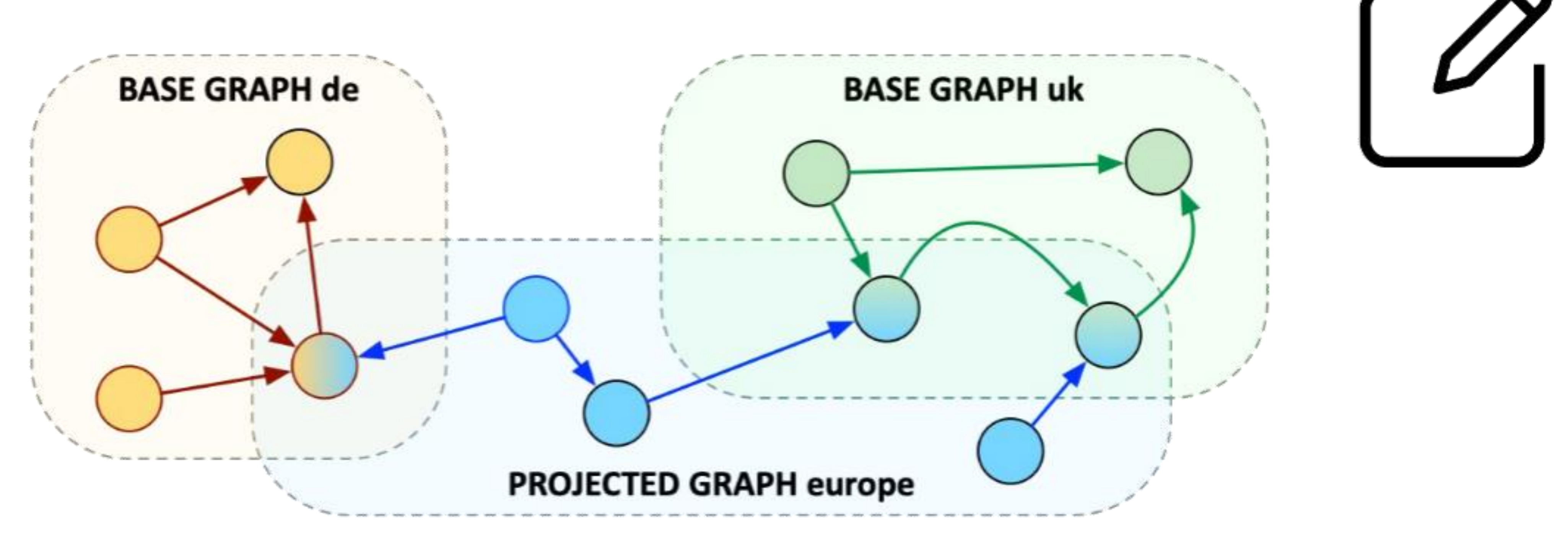
BNE-023: Advanced Types

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]

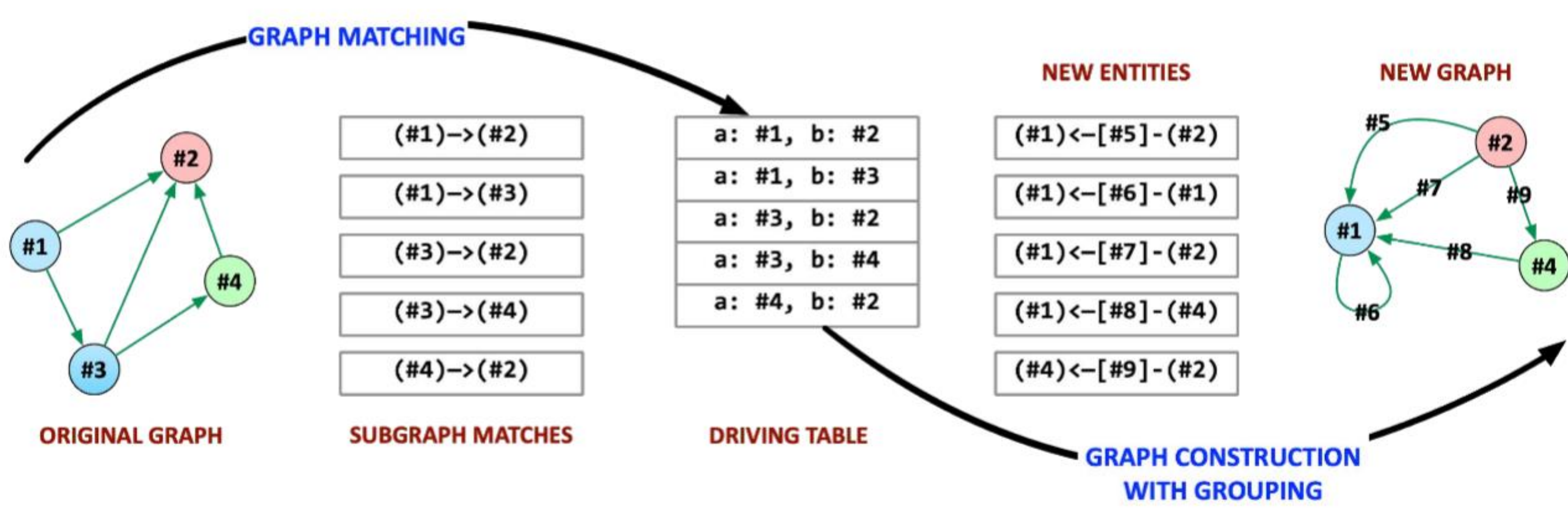


BNE-023: Graph projection

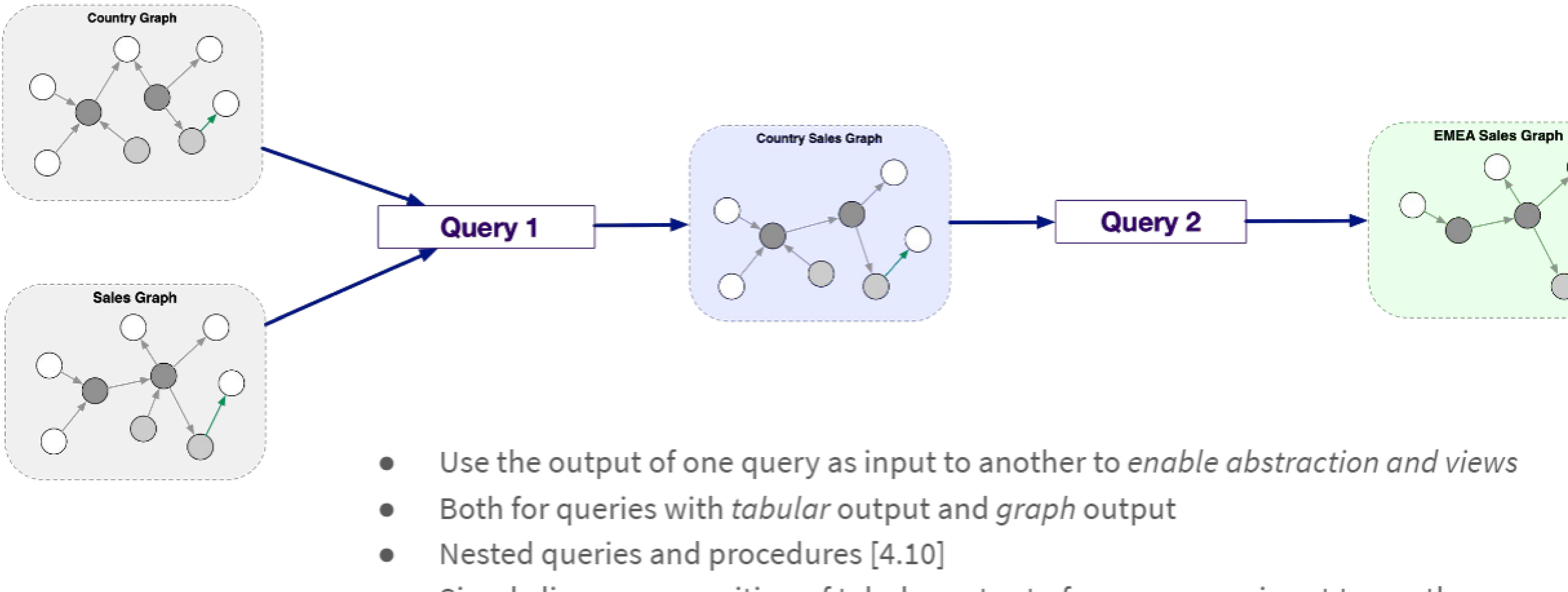


- Deriving identical elements in the projected graph ("sharing")
- Deriving new elements in the projected graph
- Shared edges always point to the same (shared) endpoints in the projected graph

BNE-023: Graph projection is inverse pattern matching

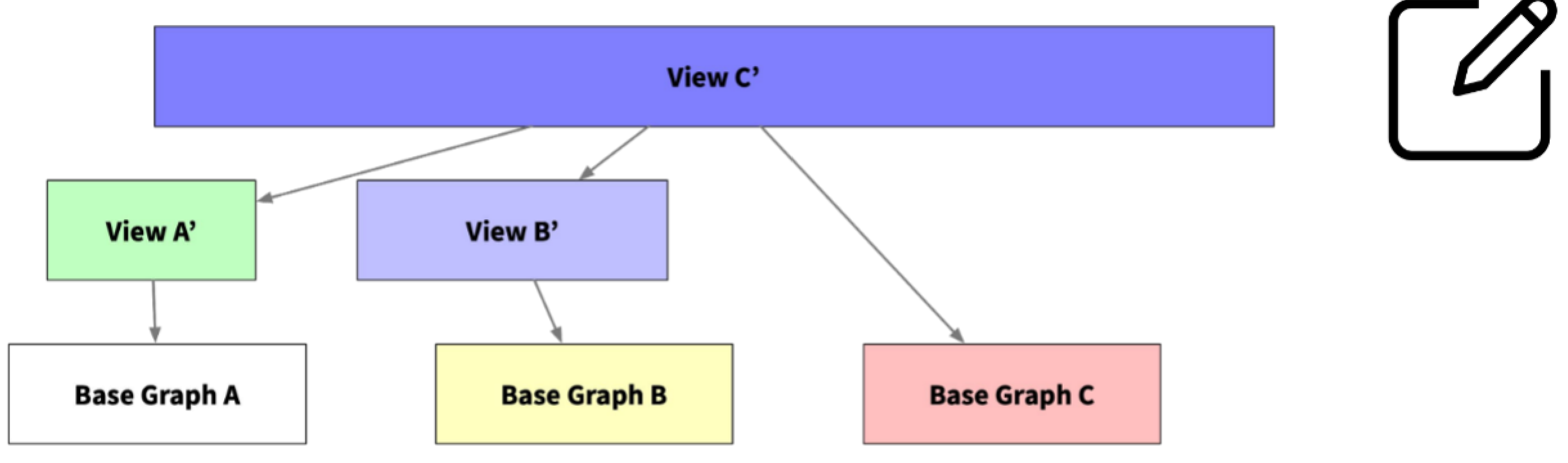


BNE-023: Queries Are Procedures [4.3]



- 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]

BNE-023: Views [3.7, 4.12]



- Graph elements in views are derived from other graphs (which may again be views)
- Graph elements are "owned" by their base graph or introducing views
- Derivation graph must form a DAG
- Updates reverse transformation

BNE-023: Views [3.7, 4.12]

- A (graph) view is a query† that returns a graph
  - GQL could also support tabular views
- A view can be used as if it was a graph
  - a tabular view can be used as if it was a table
- Queries (incl. views) can be parameterized
  - allowing the application of the same transformation over compatible graphs

```
CREATE QUERY foaf($input SocialGraph) AS {
  FROM $input
  MATCH (a)-[IS FRIEND]-()-[IS FRIEND]-(b)
  CONSTRUCT (a)-[IS FOAF]-(b)
}
```

```
FROM foaf(facebook) MATCH ...
FROM foaf(twitter) MATCH ...
```

BNE-023: Graph Augmentation

View C'

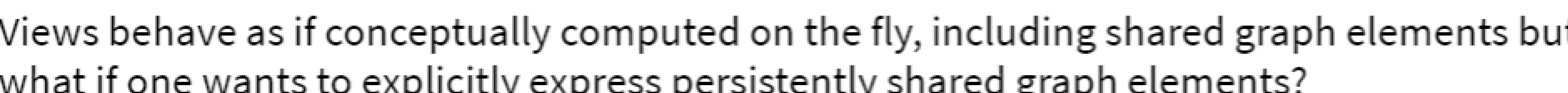
View A'

View B'

Base Graph A

Base Graph B

Base Graph C



Views behave as if conceptually computed on the fly, including shared graph elements but what if one wants to explicitly express persistently shared graph elements?

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

Many open questions (e.g. deletion semantics, security model implications)

BNE-023: GQL Scope And Features Document

A new and independent

Declarative,

Composable,

Compatible,

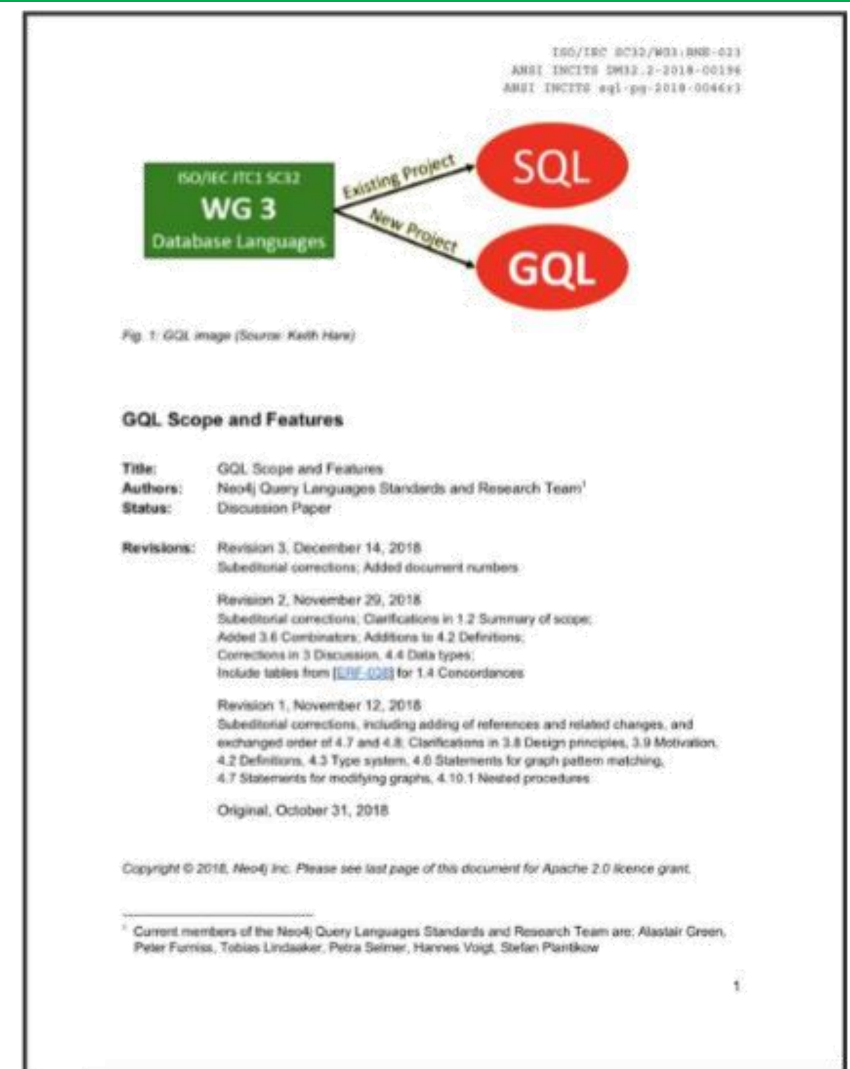
Modern,

Intuitive

Property Graph Query Language

http://tiny.cc/gql-scope-and-features

http://tiny.cc/gql-scope-digest



ISO/IEC SC32/WG3:BNE-023
ANSI INCITS DM32.2-2018-00196
ANSI INCITS sql-pg-2018-0046r3

Fig. 1: GQL image (Source: Keith Hare)

GQL Scope and Features

Title: GQL Scope and Features
Authors: Neo4j Query Languages Standards and Research Team[1]
Status: Discussion Paper

Revisions: Revision 3, December 14, 2018
Subeditorial corrections; Added document numbers

Revision 2, November 29, 2018
Subeditorial corrections; Clarifications in 1.2 Summary of scope;
Added 3.6 Combinators; Additions to 4.2 Definitions;
Corrections in 3 Discussion, 4.4 Data types;
Include tables from [ERF-028] for 1.4 Concordances

Revision 1, November 12, 2018
Subeditorial corrections, including adding of references and related changes, and exchanged order of 4.7 and 4.8; Clarifications in 3.8 Design principles, 3.9 Motivation, 4.2 Definitions, 4.3 Type system, 4.6 Statements for graph pattern matching, 4.7 Statements for modifying graphs, 4.10.1 Nested procedures

Original, October 31, 2018

Copyright © 2018, Neo4j Inc. Please see last page of this document for Apache 2.0 licence grant.

[1] Current members of the Neo4j Query Languages Standards and Research Team are: Alastair Green, Peter Furniss, Tobias Lindaaker, Petra Selmer, Hannes Voigt, Stefan Plantikow

1

Bibliography

- Foundations of Modern Query Languages for Graph Databases. R. Angles, M. Arenas, P. Barcelo, A. Hogan, J. Reutter, D. Vrgoc. ACM Computing Surveys 50, 5 (2017). marenas.sitios.ing.uc.cl/publications/csur17.pdf
- Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries Towards Understanding Modern Graph Processing, Storage, and Analytics. M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstwski, C. Barthels, G. Alonso, T. Hoefler (2019). arxiv.org/pdf/1910.09017.pdf
- G: A graphical query language supporting recursion. I. Cruz, A. Mendelzon, P. Wood. SIGMOD (1987). dl.acm.org/doi/pdf/10.1145/38714.38749
- Cypher: An Evolving Query Language for Property Graphs. N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V Marsault, S. Plantikow, M. Rydberg, P. Selmer, A. Taylor. SIGMOD (2018). homepages.inf.ed.ac.uk/libkin/papers/sigmod18.pdf
- The Gremlin Graph Traversal Machine and Language. M. Rodriguez (2015). arxiv.org/pdf/1508.03843.pdf
- TigerGraph: A Native MPP Graph Database. A. Deutsch, Y. Xu, M. Wu, V. Lee (2019). arxiv.org/pdf/1901.08248.pdf
- PGQL: a Property Graph Query Language. O. van Rest, S. Hong, J. Kim, X. Meng, H.Chafi. GRADES (2016). event.cwi.nl/grades/2016/07-VanRest.pdf
- G-CORE: A Core for Future Graph Query Languages Designed by the LDBC Graph – designed by the LDBC Query Language Task Force. R. Angles, M. Arenas, P. Barcelo, P. Boncz, G. Fletcher, C. Guttierrez, T. Lindaaler, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, H. Voigt. SIGMOD (2018). arxiv.org/pdf/1712.01550.pdf
- GQL Scope and Features. – by the Neo4j Query Languages Standards and Research Team. A. Green, P. Furniss, T. Lindaaker, P. Selmer, H. Voigt, S. Plantikow (2018). tiny.cc/gql-scope-and-features