The graph & RDF benchmark reference

G-CORE:

A Core for Future Graph Query Languages

LDBC GraphQL task force, including Peter Boncz (CWI)

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

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

Where does G-CORE come from?

- This work is the culmination of 2.5 years of intensive discussion between LDBC and **industry**, including:
  - Capsenta, HP, Huawei, IBM, Neo4j Oracle, SAP and Sparsity.

Application Fields	
healthcare / pharma	14
publishing	10
finance / insurance	6
cultural heritage	6
e-commerce	5
social media	4
telecommunications	4

Used Features	
graph reachability	36
graph construction	34
pattern matching	32
shortest path search	19
graph clustering	14

Figure 1: Graph database usage characteristics derived from the use-case presentations in LDBC TUC Meetings 2012-2017 (source: https://github.com/ldbc/tuc_presentations).

- The **Graph Query Language Task Force** designed this language.
  - members combine strong expertise in theory, systems and products
  - led by Marcelo Arenas.

LDBC

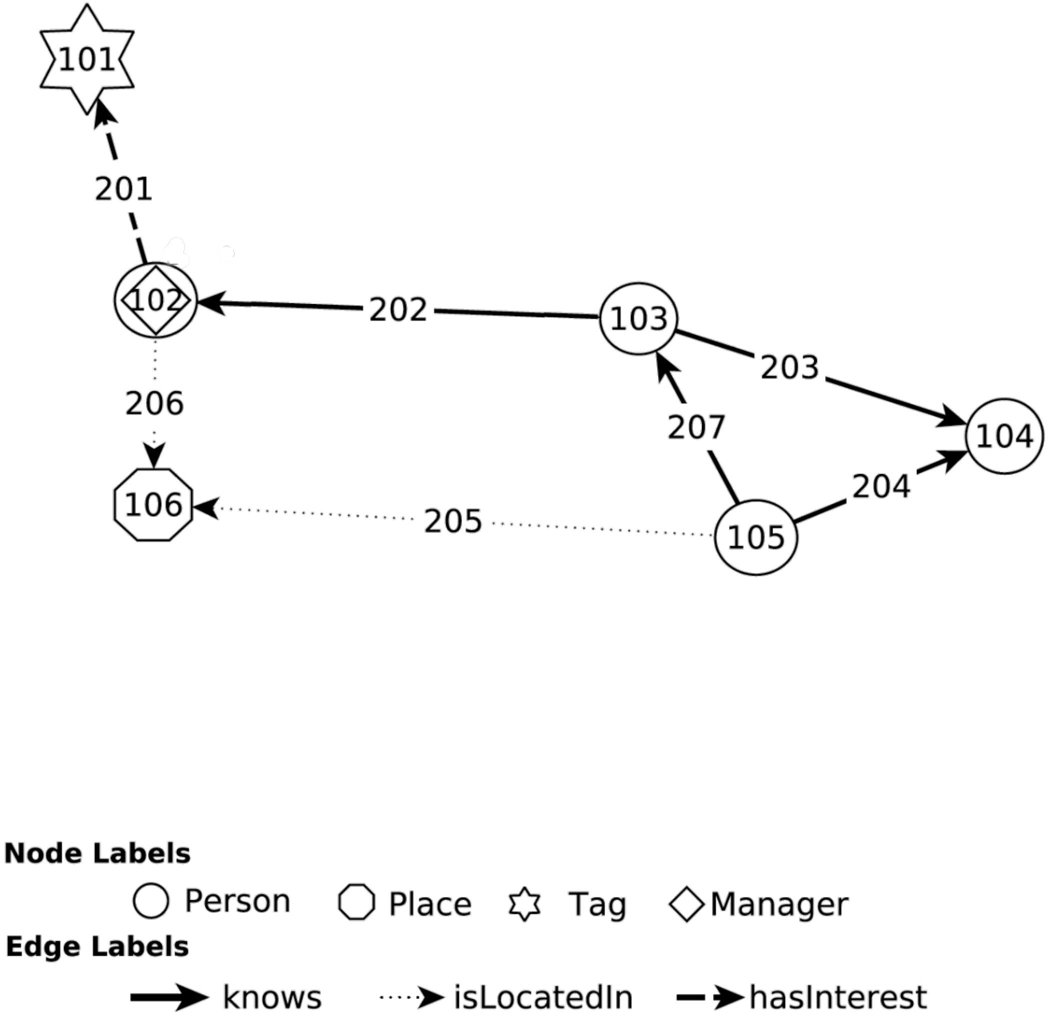
LDBC Graph Query Language Task Force

- 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 (→DLR)
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

LDBC

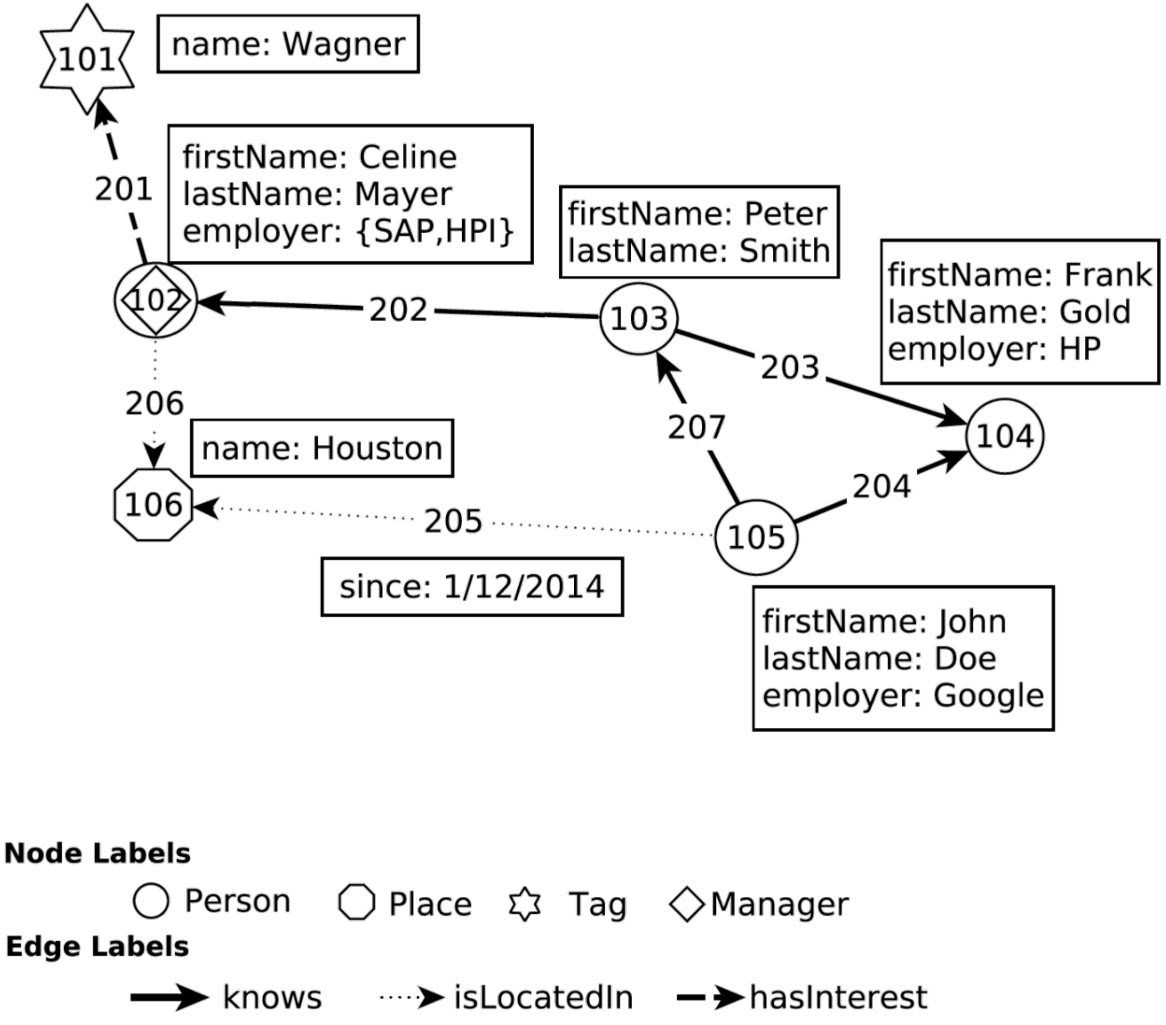
Graph Data Model



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

Example from **SNB**:
LDBC Social Network Benchmark
(see SIGMOD 2015 paper)

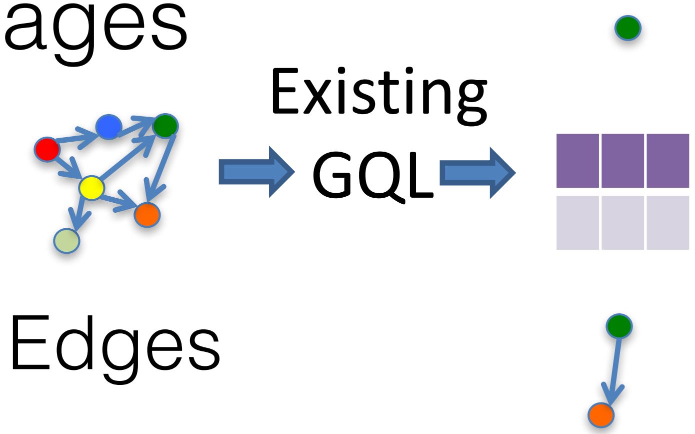
Property Graph Data Model



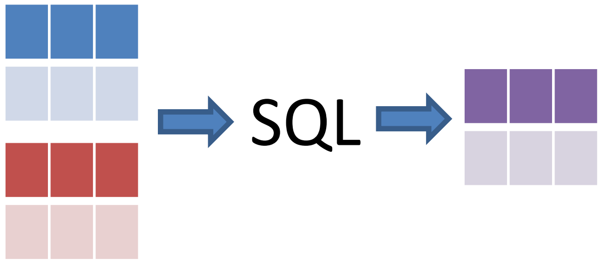
- **directed** graph
- nodes & edges are **entities**
- entities can have **labels**
- ..and (**property,value**) pairs

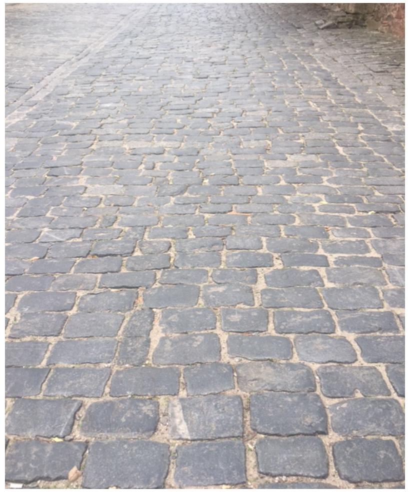
CHALLENGE 1: COMPOSABILITY

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



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





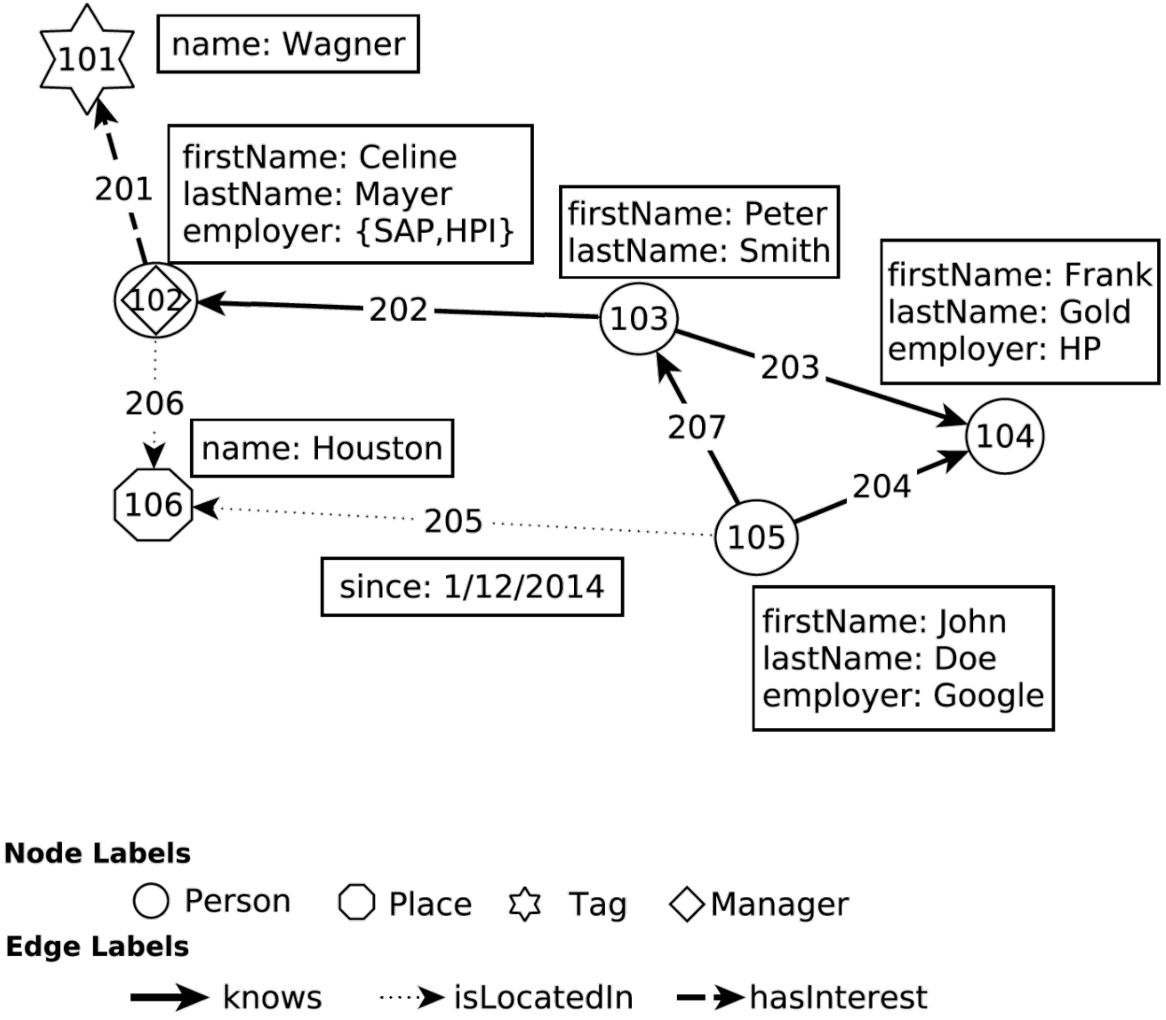
LDBC

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)

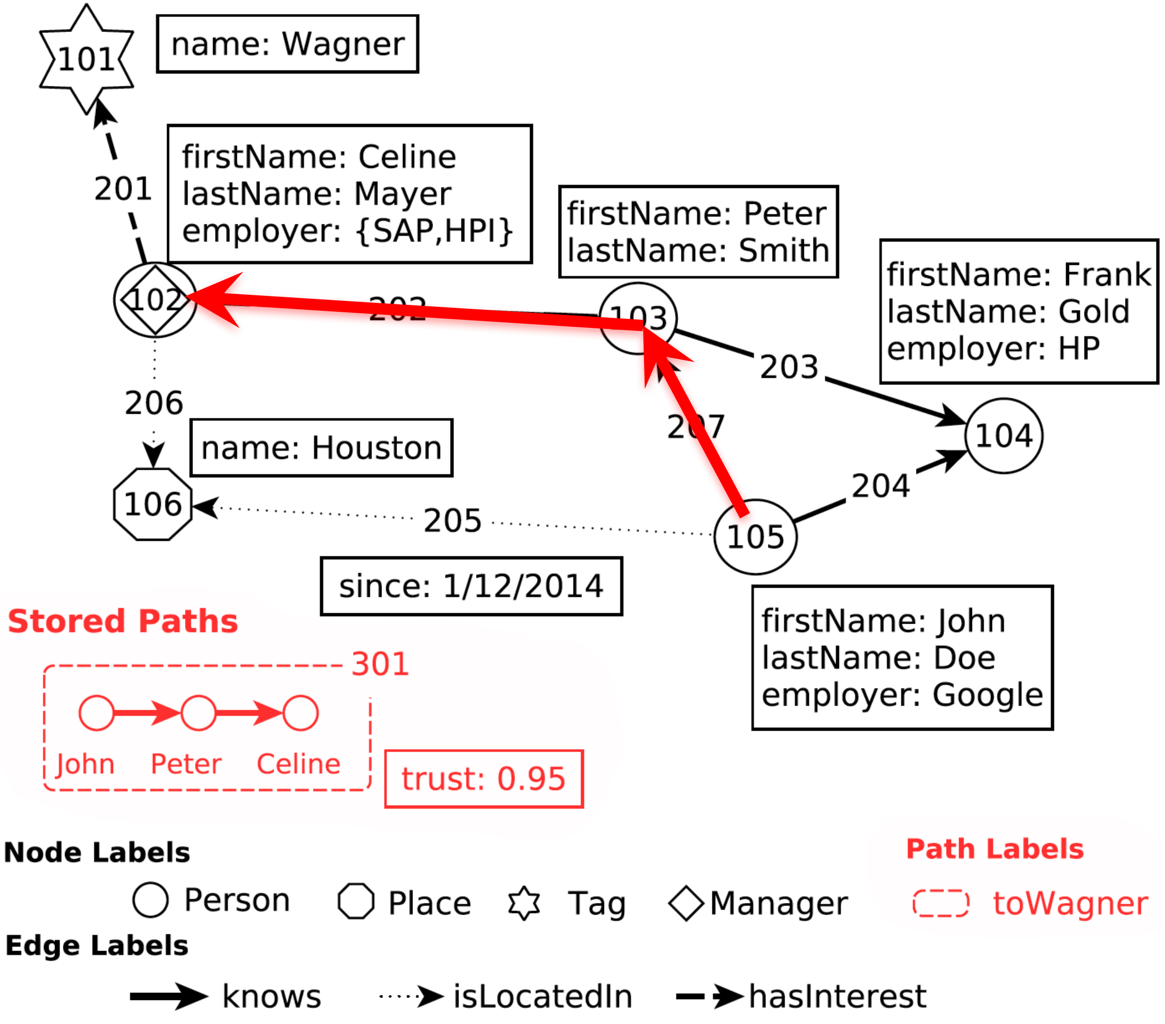
- Why is it important?
  - Paths are fundamental to Graphs
  - Increase the expressivity of the language; do more within the language

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

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..

- 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
```

- 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

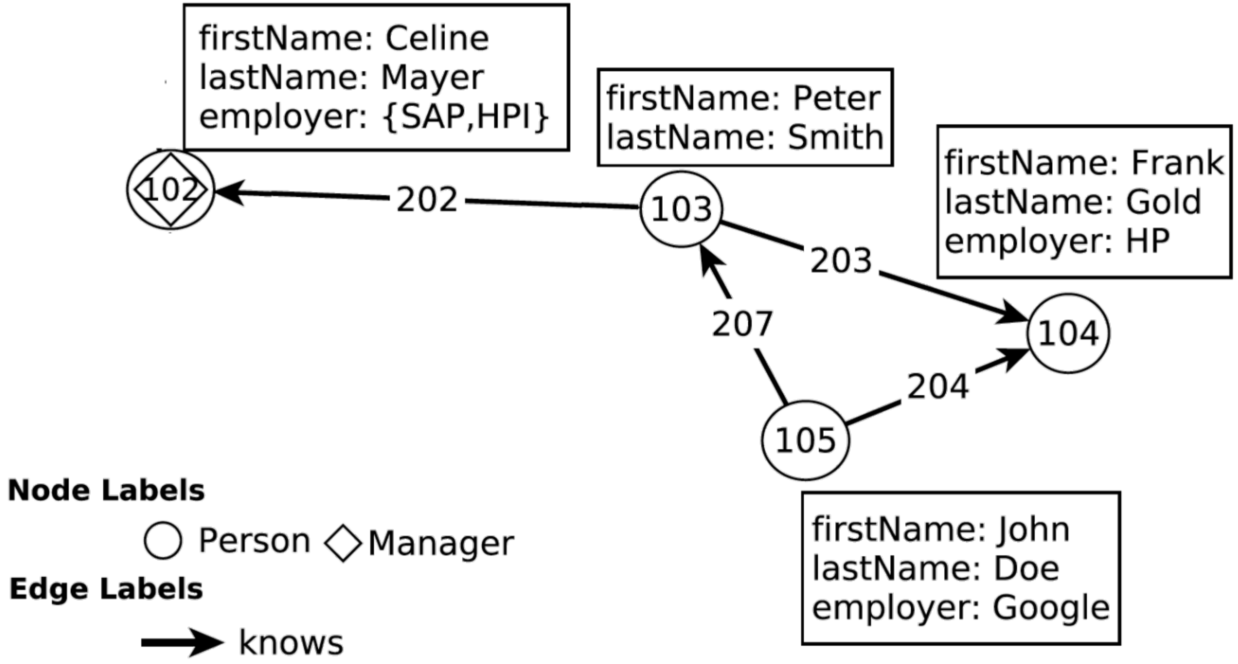
n
105 #John
104 #Frank
103 #Peter
102 #Celine

LDBC

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
```

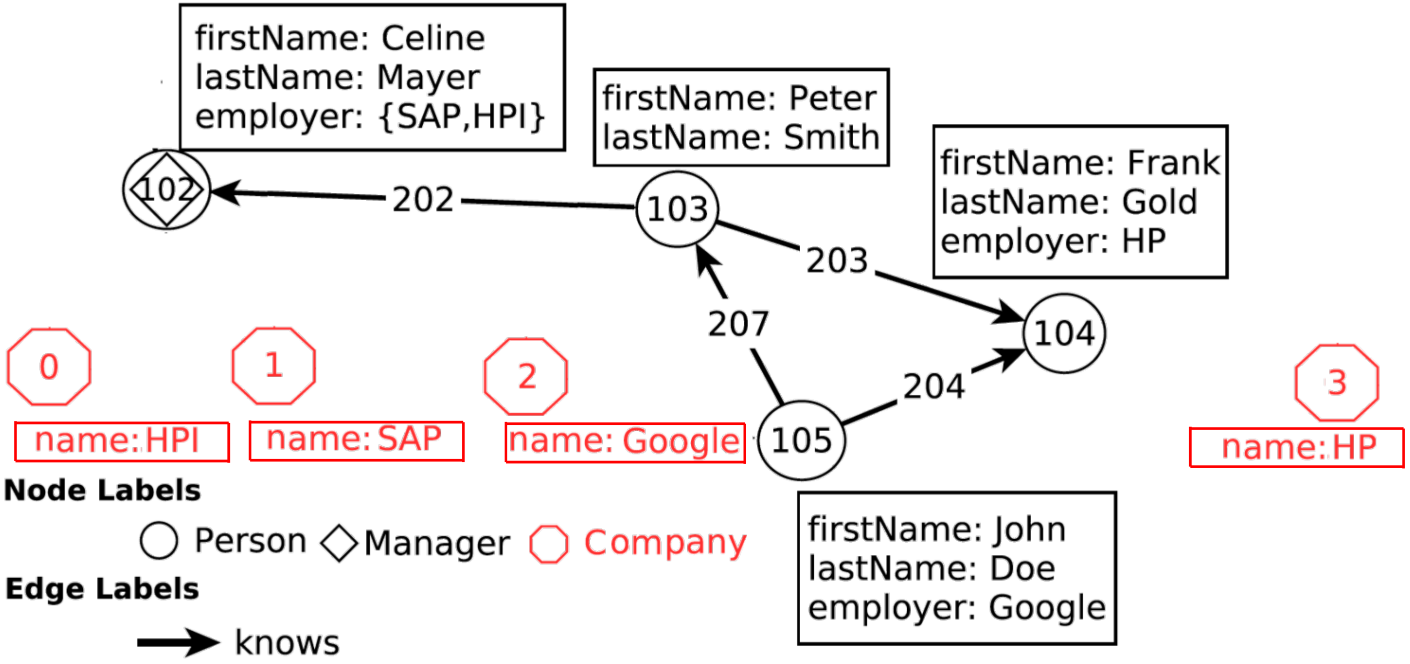
c	n
0 #Google	105 #John
1 #HPI	104 #Frank
2 #SAP	102 #Celine
3 #HP	102 #Celine



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
```

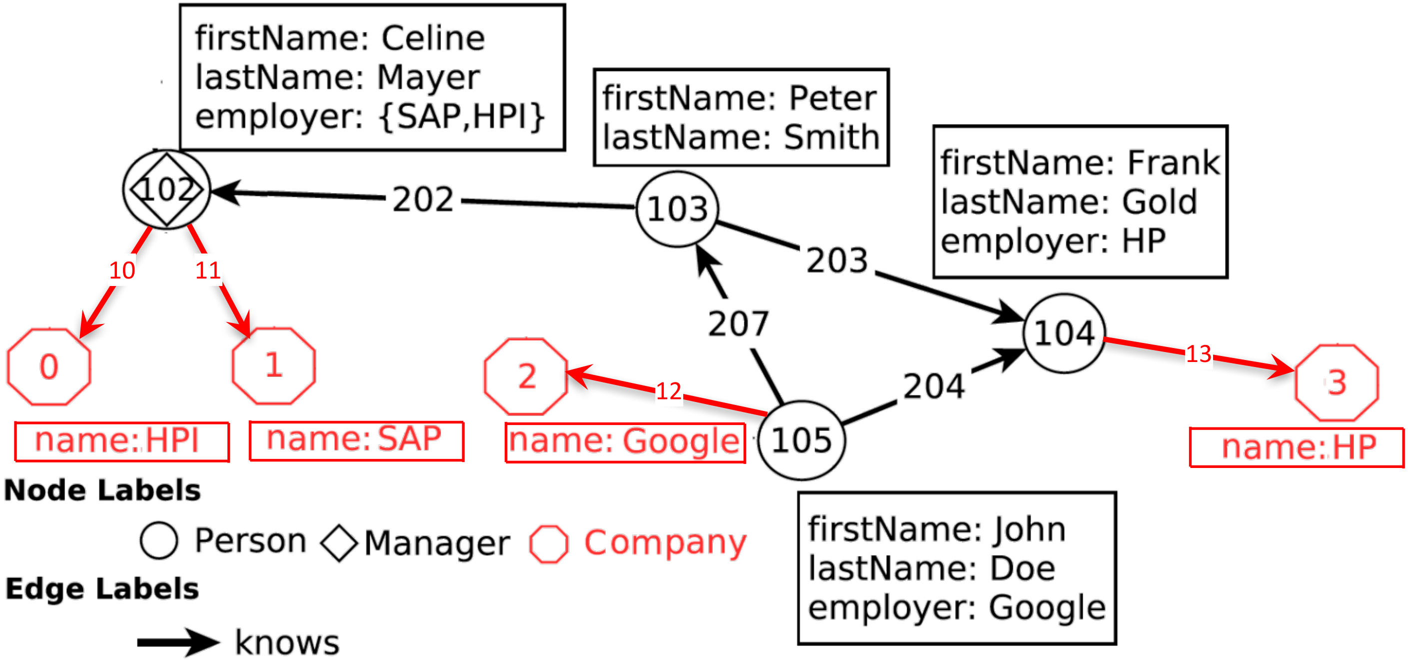
c	n
0 #Google	105 #John
1 #HPI	104 #Frank
2 #SAP	102 #Celine
3 #HP	102 #Celine



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

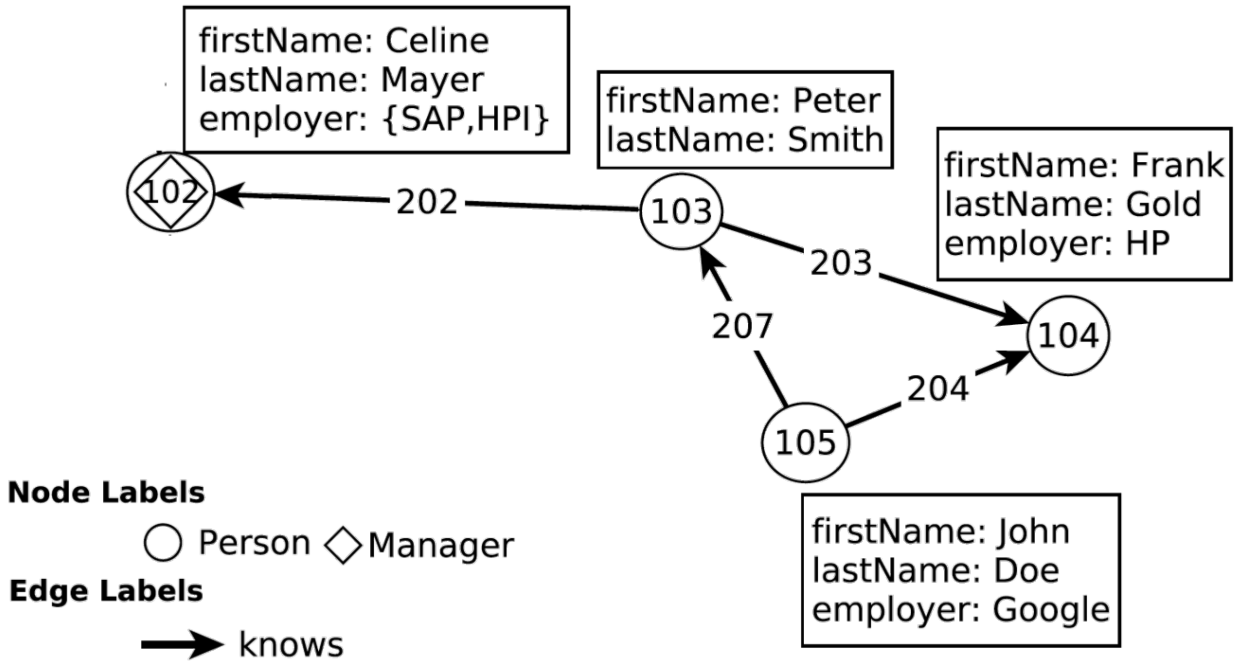
```
CONSTRUCT social_graph,
 (n)-[y:worksAt]->(x GROUP e :Company {name=e})
MATCH (n:Person {employer=e}) ON social_graph
```

- 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

LDBC

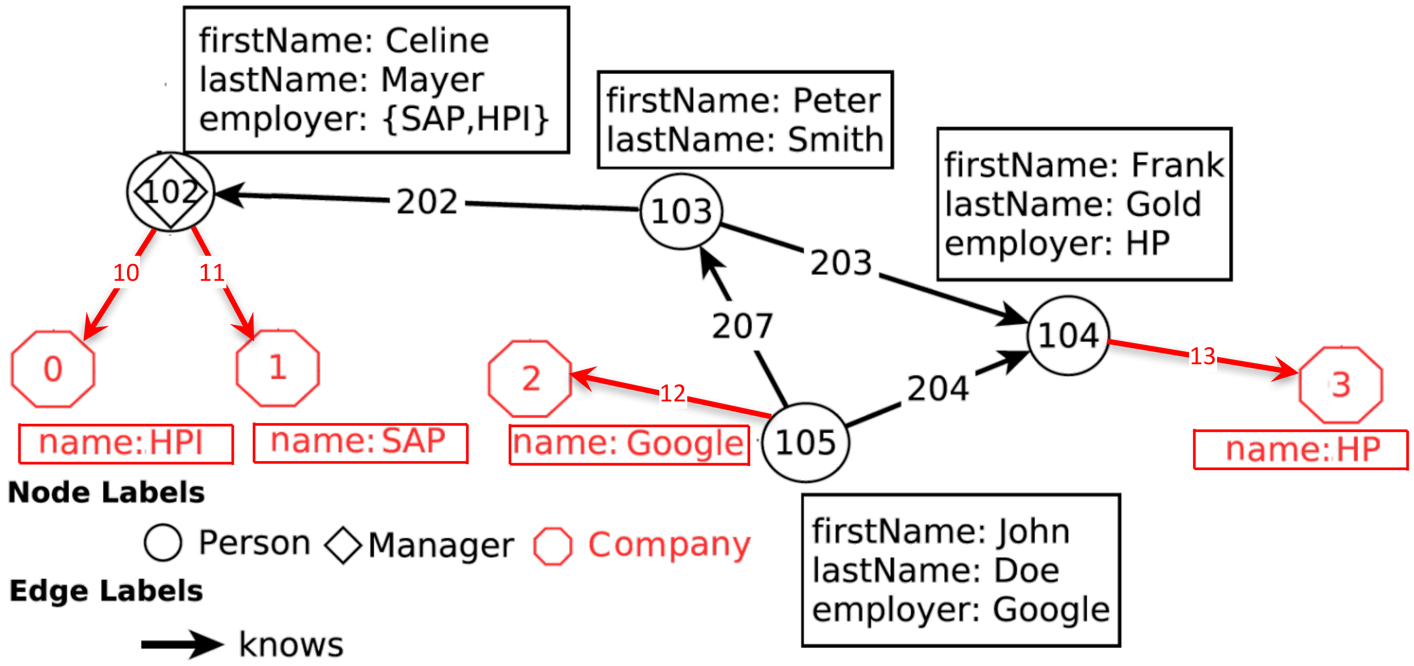
Creating Graphs from Values

```
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

LDBC

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)
      )
```

LDBC

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)

LDBC

More G-CORE..

More features: most advanced GQL so far. Read the 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
```

LDBC

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
```

LDBC

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
```

LDBC

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
```

LDBC

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 **`SELECT`** and **`FROM.`** combine with Cartesian Product, as usual.

Result:

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

Take-Away

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