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.

<table>
<thead>
<tr>
<th>Application Fields</th>
<th>Used Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>healthcare / pharma</td>
<td>graph reachability</td>
</tr>
<tr>
<td>publishing</td>
<td>graph construction</td>
</tr>
<tr>
<td>finance / insurance</td>
<td>pattern matching</td>
</tr>
<tr>
<td>cultural heritage</td>
<td>shortest path search</td>
</tr>
<tr>
<td>e-commerce</td>
<td>graph clustering</td>
</tr>
<tr>
<td>social media</td>
<td></td>
</tr>
<tr>
<td>telecommunications</td>
<td></td>
</tr>
</tbody>
</table>

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.
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

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

**Node Labels**

- Person
- Place
- Tag
- Manager

**Edge Labels**

- knows
- locatedIn
- hasInterest
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

Existing

GQL

SQL
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

**Node Labels**
- Person
- Place
- Tag
- Manager

**Edge Labels**
- knows
- isLocatedIn
- hasInterest
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.

<table>
<thead>
<tr>
<th>c</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#Google</td>
</tr>
<tr>
<td>1</td>
<td>#HPI</td>
</tr>
<tr>
<td>2</td>
<td>#SAP</td>
</tr>
<tr>
<td>3</td>
<td>#HP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>105</td>
</tr>
<tr>
<td>104</td>
</tr>
<tr>
<td>102</td>
</tr>
<tr>
<td>102</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#HPI</td>
</tr>
<tr>
<td>1</td>
<td>#SAP</td>
</tr>
<tr>
<td>2</td>
<td>#Google</td>
</tr>
<tr>
<td>3</td>
<td>#HP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>105</td>
</tr>
<tr>
<td>104</td>
</tr>
<tr>
<td>103</td>
</tr>
<tr>
<td>102</td>
</tr>
</tbody>
</table>
Multi-Graph Queries and Joins

**CONSTRUCT**  \( (c) \leftarrow [:\text{worksAt}] - (n) \)

**MATCH**  \( (c:\text{Company}) \) ON company_graph,

\( (n:\text{Person}) \) ON social_graph

**WHERE**  \( c.\text{name} = n.\text{employer} \)

**UNION**  social_graph

<table>
<thead>
<tr>
<th>c</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#Google 105 #John</td>
</tr>
<tr>
<td>1</td>
<td>#HPI 104 #Frank</td>
</tr>
<tr>
<td>2</td>
<td>#SAP 102 #Celine</td>
</tr>
<tr>
<td>3</td>
<td>#HP 102 #Celine</td>
</tr>
</tbody>
</table>
Multi-Graph Queries and Joins

**CONSTRUCT** \((c) \leftarrow [:\text{worksAt}] - (n)\)

**MATCH** \((c: \text{Company}) \text{ ON } \text{company\_graph,} \quad (n: \text{Person}) \text{ ON } \text{social\_graph}\)

**WHERE** \(c.\text{name} = n.\text{employer}\)

**UNION** \(\text{social\_graph}\)
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
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 ...
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
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)

WHERE ...

EXISTS (

    CONSTRUCT ()

    MATCH (n)-[:isLocatedIn]->()-[:isLocatedIn]-(m)
)

Syntactical shorthand for existential subquery:
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 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
```
• views

```graphql
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

```sql
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

```sql
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
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 `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
   \[ 1+2 = \text{the data model of G-CORE is graphs-with-paths (PPG)} \]

- 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