The (sorry) State of Graph Database Systems

Peter Boncz
CWI

comparing graph with relational database systems..

+ provide pointers to related literature
About Myself

- **Systems**
  - Column stores (MonetDB)
  - Vectorized execution, Lightweight compression (Actian Vector/VectorWise)

- **Benchmarking**
  - LDBC: Linked Data Benchmark Council (ldbcouncil.org)
    - Social Network Benchmark (Interactive / BI)
    - Graphalytics

- **Query Languages**
  - G-CORE - with e.g. Neo4j, Oracle, and researchers from the theory community
    - LDBC Liaison with ISO ⇒ SQL:2023 (SQL/PGQ)
Roadmap

● above the surface: **Graph Data Management**
  ○ data models
  ○ query languages
  ○ systems

● under the hood: **Graph Systems**
  ○ 6 blunders in graph system architecture
  ○ blueprint of a competent graph database system
  ○ future standards: SQL/PGQ (SQL:2023) and GQL
Graph Data Management
GDBMS Use Cases

Gained a foothold in the data systems market

- Initially via RDF and SPARQL systems
- now via Property Graph Systems

Tasks: Data Integration, Data cleaning and Enrichment, Fraud Detection, Recommendation, Historical Analysis, Root-Cause Analysis,...

Data: knowledge graphs, social networks, telco networks, relational warehouses, data lakes (output of joins, similarity mining generates edges on-the-fly)

The Future Is Big Graphs: A Community View on Graph Processing Systems


Contributed Articles

Communications of the ACM, September 2021, Vol. 64, No. 9, Pages 62-71
Key GDBMS building blocks

- Property graph data model
- Graph query language
- Graph visualization
- Subgraph matching
- Relational queries
- Path queries
- Stored procedures
Data model: Property graph

Directed graph consisting of labeled entities: vertexes & edges

- Entities can have properties with (literal) values (KV-pairs)
- Loose schema only

SUBCLASS_OF

Tag

LIKES

Person

MEMBER

Forum

KNOWS

Tag

name: Oasis

LIKES

Forum

MEMBER

since: 2017-05-03

Person

name: Alice

speaks: [en, fr]
**SIGMOD’21**

**PG-KEYS: Keys for Property Graphs**

- Renzo Angles, Universidad de Talca, IMFiD Chile
- Angela Bonifati, Lyon 1 Univ., Liris CNRS & INRIA
- Stefania Dumbrava, ENSIEE & Inst. Polytechnique de Paris
- George Fletcher, Eindhoven Univ. of Technology
- Keith W. Hare, JCC Consulting Inc., Neo4j
- Jan Hidders, Birkbeck, Univ. of London
- Victor E. Lee, TigerGraph
- Beï Li, Google LLC
- Leonid Libkin, U. of Edinburgh, ENS-Paris/PSL, Neo4j
- Wim Martens, University of Bayreuth
- Filip Murlak, University of Warsaw
- Josh Perryman, Interos Inc.
- Ognjen Savkovic, Free Univ. of Bozen-Bolzano
- Michael Schmidt, Amazon Web Services
- Juan Sequeda, data.world
- Sławek Staworko, U. Lille, INRIA LINKS, CRISTAL CNRS
- Dominik Tomaszuk, Inst. of Comp. Sci., U. of Białystok

**ABSTRACT**

We report on a community effort between industry and academia to shape the future of property graph constraints. The standardization for a property graph query language is currently underway through the ISO Graph Query Language (GQL) project. Our position is that this project should pay close attention to schemas and constraints, and should focus next on key constraints.

The main purposes of keys are enforcing data integrity and allowing the referencing and identifying of objects. Motivated by use cases from our industry partners, we argue that key constraints

**KEYWORDS**

- property graphs
- key constraints

**ACM Reference Format:**

Reconciliation of RDF* and Property Graphs

Olaf Hartig
University of Waterloo
http://olafhartig.de

November 14, 2014

Abstract
Both the notion of Property Graphs (PG) and the Resource Description Framework (RDF) are commonly used models for representing graph-shaped data. While there exist some system-specific solutions to convert data from one model to the other, these solutions are not entirely compatible with one another and none of them appears to be based on a formal foundation. In fact, for the PG model, there does not even exist a commonly agreed-upon formal definition.
Data model: RDF triples

vs Property Graph
Data model: RDF triples vs Property Graph

Tag
name: Oasis
D
LIKES
since: 2004/1/1

Person
name: Alice
speaks: [en, fr]
A

reification

"Oasis"
::uri:name

"Alice"
::uri:name

::uri:likes
::uri:since
2004/1/1

::uri:speaks
"en"
"fr"
Query language: Cypher

MATCH
(p1:Person)-[:KNOWS*]-(p2:Person),
(p1)-[:MEMBER]->(f:Forum)-[:MEMBER]->(p2),
WHERE NOT (f)-[:MEMBER]->(p3)
RETURN p1, f, count(p2), count(p3)
Query language: Cypher

MATCH
(p1:Person)-[:KNOWS]-(p2:Person),
(p1)<-[[:MEMBER]-[:MEMBER]->(p2),
(p1)-[:KNOWS*]-(p3:Person)
WHERE NOT (f)-[:MEMBER]-(p3)
RETURN p1, f, count(p2), count(p3)

subgraph matching
relational operators

Person
name: Alice
speaks: [en, fr]

Forum
title: Drums

<table>
<thead>
<tr>
<th>&quot;p1&quot;</th>
<th>&quot;f&quot;</th>
<th>&quot;count(p2)&quot;</th>
<th>&quot;count(p3)&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>{B}</td>
<td>{E}</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>{A}</td>
<td>{E}</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Homomorphic semantics
Pattern matching

- basic graph pattern
- complex graph pattern
Subgraph matching (Cypher)

Category: Basic graph pattern

MATCH 
(p1:Person)<-[MEMBER]-(f:Forum)-[MEMBER]-(p2:Person),
(p1)-[KNOWS]-(p2)
WHERE p1.id < p2.id
RETURN p1, p2, f

Results:
(A, B, D)
(A, B, E)
(B, C, E)
Subgraph matching (Cypher)

Category: Basic graph pattern

MATCH
(p1:Person)<-[:MEMBER]-(f:Forum)-[:MEMBER]->(p2:Person),
(p1)-[:KNOWS]-(p2)
WHERE p1.id < p2.id
RETURN p1, p2, f

Results:
(A, B, D)
(A, B, E)
(B, C, E)
Subgraph matching (Cypher)

Category: Basic graph pattern

MATCH
  (p1:Person)<-[MEMBER]-(f:Forum)-[MEMBER]->(p2:Person),
  (p1)-[KNOWS]-(p2)
WHERE p1.id < p2.id
RETURN p1, p2, f

Results:
(A, B, D)
(A, B, E)
(B, C, E)
Subgraph matching (Cypher)

Category: **Basic graph pattern**

MATCH

(p1:Person)<-[MEMBER]-(f:Forum)-[MEMBER]-(p2:Person),
(p1)-[KNOWS]-(p2)
WHERE p1.id < p2.id
RETURN p1, p2, f

Results:

(A, B, D)
(A, B, E)
(B, C, E)
Subgraph matching (SQL)

**Edge tables:** knows(person1id, person2id); member(forumid, personid)

**Basic graph pattern:** equijoins (SPJ)

```
SELECT m1.personid, m2.personid, m1.forumid
FROM member m1
JOIN member m2
    ON m1.forumid = m2.forumid
JOIN knows
    ON knows.person1id = m1.personid
    AND knows.person2id = m2.personid
WHERE knows.person1id < knows.person2id
```
Pattern matching

- basic graph pattern
- complex graph pattern
Subgraph matching (Cypher)

Category: **Complex graph pattern**

MATCH (f:Forum)-[:MEMBER]->(p1:Person)
OPTIONAL MATCH (f)-[:MEMBER]->(p2:Person)
WHERE p1.id < p2.id AND NOT (p1)-[:KNOWS]-(p2)
RETURN f, count(p2)

Results:

(D, 0)
(E, 1)
Subgraph matching (SQL)

**Edge tables:** knows(person1id, person2id); member(forumid, personid)

**Complex graph pattern:** equijoins, outer joins, antijoin, aggregation (SPOJG)

```sql
SELECT m1.forumid, count(m2.personid)
FROM member m1
LEFT OUTER JOIN member m2
ON m1.forumid = m2.forumid
AND m1.personid < m2.personid
WHERE NOT EXISTS (SELECT true FROM knows
WHERE person1id = m1.personid
AND person2id = m2.personid)
GROUP BY m1.forumid
```
Unweighted shortest path in Cypher

MATCH path=shortestPath(
  (source:Person {name: 'Bob'})-[[:KNOWS*]]-(target:Person {name: 'Fleur'})
)
RETURN length(path) AS length

Result:

length

3
Unw. SP query: Data in SQL

Graphs can be represented in the relational model with PKs and FKs (primary keys and foreign keys)

Person

<table>
<thead>
<tr>
<th>id [PK]</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
</tr>
<tr>
<td>3</td>
<td>Cecile</td>
</tr>
<tr>
<td>4</td>
<td>Diane</td>
</tr>
<tr>
<td>5</td>
<td>Emily</td>
</tr>
<tr>
<td>6</td>
<td>Fleur</td>
</tr>
</tbody>
</table>

knows

<table>
<thead>
<tr>
<th>person1id [FK]</th>
<th>person2id [FK]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

all edges backwards
Graphs can be represented in the relational model with PKs and FKs (primary keys and foreign keys).

### Unw. SP query: Data in SQL

#### Person

<table>
<thead>
<tr>
<th>id [PK]</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
</tr>
<tr>
<td>3</td>
<td>Cecile</td>
</tr>
<tr>
<td>4</td>
<td>Diane</td>
</tr>
<tr>
<td>5</td>
<td>Emily</td>
</tr>
<tr>
<td>6</td>
<td>Fleur</td>
</tr>
</tbody>
</table>

#### knows

<table>
<thead>
<tr>
<th>person1id [FK]</th>
<th>person2id [FK]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

all edges backwards
Unweighted shortest path query in SQL

WITH RECURSIVE paths(source, target, path, level, targetReached) AS (  
  SELECT person1id AS source,  
         person2id AS target,  
         [person1id, person2id] AS path,  
         1 AS level,  
         (p2.name = 'Fleur') AS targetReached  
  FROM knows  
  JOIN Person p1 ON p1.id = knows.person1id  
  JOIN Person p2 ON p2.id = knows.person2id  
  WHERE p1.name = 'Bob'  
UNION ALL  
  SELECT paths.source AS source,  
         person2id AS target,  
         array_append(path, person2id) AS path,  
         level + 1 AS level,  
         max(CASE WHEN p2.name = 'Fleur' THEN true ELSE false END)  
          OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS targetReached  
  FROM paths  
  JOIN knows ON knows.person1id = paths.target  
  JOIN Person p2 ON p2.id = knows.person2id  
  WHERE person2id != ALL(paths.path)  
          AND NOT paths.targetReached  
          AND NOT EXISTS (SELECT 1 FROM paths previous_paths WHERE list_contains(previous_paths.path, knows.person2id))  
)  
SELECT path, level, targetReached  
FROM paths  
JOIN Person ON Person.id = paths.target;
Unweighted shortest path query in SQL

WITH RECURSIVE paths(source, target, path, level, targetReached) AS (  SELECT person1id AS source,  person2id AS target,  [person1id, person2id] AS path,  1 AS level,  (p2.name = 'Fleur') AS targetReached  FROM knows  JOIN Person p1 ON p1.id = knows.person1id  JOIN Person p2 ON p2.id = knows.person2id  WHERE p1.name = 'Bob'  UNION ALL  SELECT paths.source AS source,  person2id AS target,  array_append(path, person2id) AS path,  level + 1 AS level,  max(CASE WHEN p2.name = 'Fleur' THEN true ELSE false END) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS targetReached  FROM paths  JOIN knows ON knows.person1id = paths.target  JOIN Person p2 ON p2.id = knows.person2id  WHERE person2id != ALL(paths.path)  AND NOT paths.targetReached  AND NOT EXISTS (SELECT 1 FROM paths previous_paths WHERE list_contains(previous_paths.path, knows.person2id))  )  SELECT path, level, targetReached  FROM paths  JOIN Person ON Person.id = paths.target;
Unweighted shortest path query in SQL

WITH RECURSIVE paths(source, target, path, level, targetReached) AS (
    SELECT person1id AS source,
           person2id AS target,
           [person1id, person2id] AS path,
           1 AS level,
           (p2.name = 'Fleur') AS targetReached
    FROM knows
    JOIN Person p1 ON p1.id = knows.person1id
    JOIN Person p2 ON p2.id = knows.person2id
    WHERE p1.name = 'Bob'
UNION ALL
    SELECT paths.source AS source,
           person2id AS target,
           array_append(path, person2id) AS path,
           level + 1 AS level,
           max(CASE WHEN p2.name = 'Fleur' THEN true ELSE false END)
           OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS targetReached
    FROM paths
    JOIN knows ON knows.person1id = paths.target
    JOIN Person p2 ON p2.id = knows.person2id
    WHERE person2id != ALL(paths.path)
    AND NOT paths.targetReached
    AND NOT EXISTS (SELECT 1 FROM paths previous_paths WHERE list_contains(previous_paths.path, knows.person2id))
)
SELECT array_agg(pathPerson.name) AS pathNames
FROM (SELECT path, unnest(paths.path) AS personid
      FROM paths JOIN Person targetPerson ON targetPerson.id = paths.target
      WHERE targetPerson.name = 'Fleur') unnestedPath
JOIN Person pathPerson ON pathPerson.id = unnestedPath.personid
GROUP BY path;
WITH RECURSIVE paths(startPerson, endPerson, path, level, endPersonReached) AS (  
  SELECT person1id AS startPerson, person2id AS endPerson,  
  [person1id, person2id]::bigint[] AS path, 1 AS level,  
  max(CASE WHEN p2.name = 'Fleur'  
    THEN true ELSE false END) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS endPersonReached  
  FROM knows  
  JOIN Person p1 ON p1.id = knows.person1id  
  JOIN Person p2 ON p2.id = knows.person2id  
  WHERE p1.name = 'Bob'  
  UNION ALL  
  SELECT paths.startPerson AS startPerson, person2id AS endPerson,  
  array_append(path, person2id) AS path, level + 1 AS level,  
  max(CASE WHEN p2.name = 'Fleur'  
    THEN true ELSE false END) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS endPersonReached  
  FROM paths  
  JOIN knows ON paths.endPerson = knows.person1id  
  JOIN Person p2 ON p2.id = knows.person2id  
  WHERE p2.id != ALL(paths.path)  
  AND NOT paths.endPersonReached  
)  
SELECT path, level, endPersonReached AS epr  
FROM paths;
Unweighted shortest path in Cypher

MATCH p=shortestPath(
  (start:Person {name: 'Bob'})-[[:KNOWS*]]-(end:Person {name: 'Fleur'}))
RETURN length(p) AS length

Result:

length
------
3
Graphs can be represented in the relational model with PKs and FKS (primary keys and foreign keys)

Person

<table>
<thead>
<tr>
<th>id [PK]</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
</tr>
<tr>
<td>3</td>
<td>Cecile</td>
</tr>
<tr>
<td>4</td>
<td>Diane</td>
</tr>
<tr>
<td>5</td>
<td>Emily</td>
</tr>
<tr>
<td>6</td>
<td>Fleur</td>
</tr>
</tbody>
</table>

knows

<table>
<thead>
<tr>
<th>person1id [FK]</th>
<th>person2id [FK]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

all edges backwards (optional)
Unw. SP query: Data in SQL

Graphs can be represented in the relational model with PKs and FKs (primary keys and foreign keys)
Unweighted shortest path query in SQL

WITH RECURSIVE paths(startPerson, endPerson, path, level, endPersonReached) AS (
    SELECT person1id AS startPerson, person2id AS endPerson,
    [person1id, person2id]::bigint[] AS path,
    1 AS level,
    max(CASE WHEN p2.name = 'Fleur' THEN true ELSE false END) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS endPersonReached
    FROM knows
    JOIN Person p1 ON p1.id = knows.person1id
    JOIN Person p2 ON p2.id = knows.person2id
    WHERE p1.name = 'Bob'
    UNION ALL
    SELECT paths.startPerson AS startPerson, person2id AS endPerson,
    array_append(path, person2id) AS path,
    level + 1 AS level,
    max(CASE WHEN p2.name = 'Fleur' THEN true ELSE false END) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS endPersonReached
    FROM paths
    JOIN knows ON paths.endPerson = knows.person1id
    JOIN Person p2 ON p2.id = knows.person2id
    WHERE p2.id != ALL(paths.path) AND NOT paths.endPersonReached
)
SELECT path, level, endPersonReached AS epr
FROM paths;
WITH RECURSIVE paths(startPerson, endPerson, path, level, endPersonReached) AS 
(SELECT person1id AS startPerson, person2id AS endPerson, 
    [person1id, person2id]::<bigint[]> AS path, 1 AS level, 
    max(CASE WHEN p2.name = 'Fleur' THEN true ELSE false END) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS endPersonReached 
FROM knows 
JOIN Person p1 ON p1.id = knows.person1id 
JOIN Person p2 ON p2.id = knows.person2id 
WHERE p1.name = 'Bob' 
UNION ALL 
SELECT paths.startPerson AS startPerson, person2id AS endPerson, 
    array_append(path, person2id) AS path, level + 1 AS level, 
    max(CASE WHEN p2.name = 'Fleur' THEN true ELSE false END) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS endPersonReached 
FROM paths 
JOIN knows ON paths.endPerson = knows.person1id 
JOIN Person p2 ON p2.id = knows.person2id 
WHERE p2.id != ALL(paths.path) 
AND NOT paths.endPersonReached) 
SELECT path, level, endPersonReached AS epr 
FROM paths;
Unweighted shortest path query in SQL

WITH RECURSIVE paths(startPerson, endPerson, path, level, endPersonReached) AS (  
    SELECT person1id AS startPerson, person2id AS endPerson,  
        [person1id, person2id]::bigint[] AS path,  
        1 AS level,  
        max(CASE WHEN p2.name = 'Fleur' THEN true ELSE false END) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS endPersonReached  
    FROM knows  
    JOIN Person p1 ON p1.id = knows.person1id  
    JOIN Person p2 ON p2.id = knows.person2id  
    WHERE p1.name = 'Bob'  
    UNION ALL  
    SELECT paths.startPerson AS startPerson, person2id AS endPerson,  
        array_append(path, person2id) AS path,  
        level + 1 AS level,  
        max(CASE WHEN p2.name = 'Fleur' THEN true ELSE false END) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS endPersonReached  
    FROM paths  
    JOIN knows ON paths.endPerson = knows.person1id  
    JOIN Person p2 ON p2.id = knows.person2id  
    WHERE p2.id != ALL(paths.path)  
    AND NOT paths.endPersonReached)

SELECT path, level  
FROM paths
JOIN Person ON Person.id = paths.endPerson  
WHERE Person.name = 'Fleur';

+ unnest + join to get the names

<table>
<thead>
<tr>
<th>path</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>[2, 1, 4, 6]</td>
<td>3</td>
</tr>
<tr>
<td>[2, 3, 5, 6]</td>
<td>3</td>
</tr>
<tr>
<td>[2, 3, 4, 6]</td>
<td>3</td>
</tr>
</tbody>
</table>
Path queries

- unweighted path query
- weighted shortest path query
Weighted shortest paths

Difficult. Alternative: stored procedures, e.g. Postgres has pgrouting and MADlib

Oracle example from: http://aprogrammerwrites.eu/?p=1391

WITH paths (node, path, cost, rnk, lev) AS (  
SELECT a.dst, a.src || ',' || a.dst, a.distance, 1, 1 FROM arcs a  
WHERE a.src = :SRC  
UNION ALL  
SELECT a.dst, p.path || ',' || a.dst, p.cost + a.distance, Rank () OVER (PARTITION BY a.dst ORDER BY p.cost + a.distance), p.lev + 1  
FROM paths p  
JOIN arcs a ON a.src = p.node AND p.rnk = 1  
)  
SEARCH DEPTH FIRST BY node  
SET line_no  
CYCLE node SET lp TO '*' DEFAULT ' '  
, paths_ranked AS (  
SELECT lev, node, path, cost, Rank () OVER (PARTITION BY node ORDER BY cost) rnk_t, lp, line_no  
FROM paths WHERE rnk = 1)  
SELECT LPad (node, 1 + 2^ (lev - 1), '.') node, lev, path, cost, lp  
FROM paths_ranked  
WHERE rnk_t = 1  
ORDER BY line_no

⚠ Complex query  ⚠ A relational simulation of Dijkstra’s algorithm
Weighted shortest paths

Cypher: No weighted shortest path construct. In Neo4j there’s the Graph Data Science lib.

```
MATCH (c1:Customer {id: $c1id}), (c2: Customer {id: $c2id})
CALL gds.shortestPath.dijkstra.stream(
  nodeProjection: 'Customer',
  relationshipProjection: 'TRANSFER',
  sourceNode: c1,
  targetNode: c2,
  relationshipWeightProperty: 'amount'
)
YIELD path, totalCost
RETURN path, totalCost
```

call stored procedure

This is confusing to users:
- Unweighted shortest path -> pattern matching
- Weighted shortest path -> stored procedure
Systems and languages

Cypher
PGX
PGQL
Datalog
GSQL

Cypher
Gremlin

A Survey of Current Property Graph Query Languages (2021) by Peter Boncz

See also:
ACM Computing Surveys 2017

Foundations of Modern Query Languages for Graph Databases

We survey foundational features underlying modern graph query languages. We first discuss two popular graph data models: edge-labelled graphs, where nodes are connected by directed, labelled edges; and property graphs, where nodes and edges can further have attributes. Next we discuss the two most fundamental graph query functionalities: graph pattern matching and navigational queries. We start with graph patterns and extend them to general graph patterns, subsequently introducing navigational patterns and extending them to general navigational queries. Finally, we discuss non-regular graph pattern matching and introduce the non-regular pattern matching query language KQML.
A simple test of Graph Data Systems
LSQB: A Large-Scale Subgraph Query Benchmark

Amine Mhedhbi  
University of Waterloo  
amine.mhedhbi@uwaterloo.ca

Matteo Lissandrini  
Aalborg University  
matteo@cs.aau.dk

Laurens Kuiper  
CWI Amsterdam  
laurens.kuiper@cwi.nl

Jack Waudby  
Newcastle University  
j.waudby2@newcastle.ac.uk

Gábor Szárnyas  
CWI Amsterdam  
gabor.szaranyak@cwi.nl

ABSTRACT

We introduce LSQB, a new large-scale subgraph query benchmark. LSQB tests the performance of database management systems on an important class of subgraph queries overlooked by existing benchmarks. Matching a labelled structural graph pattern, referred to as subgraph matching, is the focus of LSQB. In relational terms, the benchmark tests DBMSs’ join performance as a choke-point since subgraph matching is equivalent to multi-way joins between base Vertex and base Edge tables on ID attributes. The benchmark forms a bridge between relational and graph databases.

As observed in prior work [1, 3, 32], a subgraph matching query \( Q(V_G, E_G) \), which enumerates instances of \( G \) in an input graph \( G(V, E) \), is equivalent to a select-project-join query containing multi-way joins between base Vertex and base Edge tables. Therefore, provided a mapping from the graph schema to the relational schema, \textit{relational DBMSs} (RDBMSs) also support subgraph queries.
GDBMS performance for subgraph queries

- Load the data: 100M vertices, 650M edges
- Run all 9 queries one-by-one (count number of matches)
- Environment: cloud VM, 370GB RAM, 48 vCPU cores
GDBMS often still incompetent!

- **performance**
  - Slow loading speeds
  - Query speeds over magnitude slower than RDBMS

- **scalability**
  - Low datashize limit, typically << RAM
  - Little benefit from parallelism (SIMD, cores, machines)

- **reliability**
  - Loads never terminate
  - Query run out of memory or crash
  - Bugs
6 blunders in system architecture
Triple Fallacy 1: Locality Lost

Throwing all edges in one basket: a good idea?

- relational **clustered index**

<table>
<thead>
<tr>
<th>year</th>
<th>author</th>
<th>isbn</th>
</tr>
</thead>
<tbody>
<tr>
<td>1975</td>
<td>a1995</td>
<td>i1995</td>
</tr>
<tr>
<td>1996</td>
<td>a1996</td>
<td>i1996</td>
</tr>
<tr>
<td>1996</td>
<td>a1996</td>
<td>i1996</td>
</tr>
<tr>
<td>1997</td>
<td>a1997</td>
<td>i1997</td>
</tr>
</tbody>
</table>

- clustering is often **for free** with ZoneMaps

- relational **partitioned table**

<table>
<thead>
<tr>
<th>year</th>
<th>author</th>
<th>isbn</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995</td>
<td>a1995</td>
<td>i1995</td>
</tr>
<tr>
<td>1995</td>
<td>a1995</td>
<td>i1995</td>
</tr>
<tr>
<td>1996</td>
<td>a1996</td>
<td>i1996</td>
</tr>
<tr>
<td>1996</td>
<td>a1996</td>
<td>i1996</td>
</tr>
<tr>
<td>1997</td>
<td>a1997</td>
<td>i1997</td>
</tr>
<tr>
<td>1997</td>
<td>a1997</td>
<td>i1997</td>
</tr>
</tbody>
</table>

SELECT ?a ?n WHERE {
  ?b <has_author> ?a.
  ?b <in_year> "1996".
  ?b <isbn_no> ?n
}

an indexing on all 6 triple orders does not guarantee access locality (red)!!
Triple Fallacy 2: Join Jungle

- superfluous joins explode query complexity

**query has unnecessary joins**
- in a relational DB, this is scanning a record, not a join
- problem #1: joins are costly at query execution time
- problem #2: query optimization complexity is $O(3^N)$ with star patterns size $F$, exponentially worse $(3^F)$ optimization space coverage

```
book query:
SELECT ?a ?n WHERE {
  ?b <has_author> ?a.
  ?b <in_year> "1996".
  ?b <isbn_no> ?n
}
```
Triple Fallacy 3: Cardinality Crisis

- Graph joins are **harder to optimize**!

![Query graph]

- **because of structural correlations**
  - if \(?b\) has an \(<\text{isbn_no}\>)\) it’s a book, it has \(<\text{in_year}\>\) and \(<\text{has_author}\>\)
  - query optimizer estimates using the **independence assumption**
  - many joins (fallacy 2) + wrong estimates \(\Rightarrow\) performance disaster
4 Graph Uniqueness Syndrome

“so different from relational that no lessons apply”

- attitude also seen in research papers
- E.g. insist on using pointers for navigation (no buffer manager)
  - At what cost: updates? memory locality? fast scans?
  - Do you avoid joins, or just call them something different?

⇒ GDBMS should build on all techniques from RDBMS

- Buffer Manager, Transactions, Query Algebra, Statistics, Optimizer, …
- …and then add graph-specific functionality
5 A Pitfall: Key-Value APIs

- “APIs are faster than a query language”
  - Three navigation steps in social network = 1 million API calls
- “This GDBMS is pluggable and can use any KV store as backend”
  - Tell-tale signal of non-bulk API
  - Typically API even goes beyond process or machine

⇒ if you design an imperative API, make it a **bulk** one

- mentioned “Query Algebra” already..
6 Booby-Trapped Query Languages

- Bad: QL with high complexity and some optimizations
  - e.g. OWL
  - If the optimizer gets it, the query finishes, otherwise not

⇒ Query languages should only allow tractable queries, e.g.

  - Explicit syntax for reachability and (weighted) shortest path
    - Always Dijkstra, Bellman-Ford, ..
  - Restricted path expressions only
    - REM’s as proposed in Oracle PGQL (and G-CORE)
Blueprint of a competent GDBMS
Start from a competent base

- Columnar storage + lightweight compression
  - Compact storage, Fast (SIMD-friendly) scans
- Fast Query Executor
  - JIT (Umbra) or vectorized execution (DuckDB)
- Buffer Manager
  - data >> RAM (e.g. LeanStore = execute directly on SSD)
- Control over memory
  - C++, C or Rust
- Bottom-up Dynamic Programming Query Optimizer
  - Samples and hyperloglog as statistics
- Morsel-driven Parallellism
  - Atomics in shared hash tables, low-overhead queues
Structure-Aware Storage

GDBMS must know tables (vertex/edge entities) and its columns (aka properties)

- Either because there is an explicit schema
  - See work of LDBC Property Graph Schema working groups
- Or because the system learns the schema on-the-fly
  - Similar to smart JSON loading techniques
  - Only the most populated columns need efficient columnar storage
Faster Navigation

can we get $O(1)$ navigation using joins?

ideas:

- Positional access as a hash-join optimization (if keys are dense)
  - + caching of such hash tables
- Packed Memory Arrays (PMA)
  - Updatable graph-friendly (CSR) columnar data structure, see Teseo

---

**Teseo and the Analysis of Structural Dynamic Graphs**

Dean De Leo  
CWI  
dleo@cwi.nl

Peter Boncz  
CWI  
boncz@cwi.nl

**ABSTRACT**

We present Teseo, a new system for the storage and analysis of dynamic structural graphs in main-memory and the addition of transactional support. Teseo introduces a novel design based on sparse arrays, large arrays interleaved with gaps, and a fat tree, where the graph is ultimately stored. Our design contrasts with early systems for the analysis of dynamic graphs, which often lack transactional support and are anchored to a vertex table as a primary index. We claim that the vertex table implies several constraints, arguably representing the most compared system to day. On the other hand, there have been attempts to adapt existing Relational DBMSes (RDBMSes) for graph analysis [22, 33]. Upon inspection, these approaches have been shown to come short in terms of performance [48, 50], compared to systems for static graphs, while offering a somewhat more restricted abstraction model. Nowadays, single machines can process relatively large graphs [51], and, recently, for this architecture, several libraries to tackle dynamic graphs have been published [20, 35, 37, 46, 63].

---

**GRainDB: A Relational-core Graph-Relational DBMS**

Guodong Jin  
jingguodong@rue.cnu.edu.cn  
Renmin University of China  
China

Nafisa Anzum  
nanumz@uwwaterloo.ca  
University of Waterloo  
Canada

Semih Salihoglu  
semin.salihoglu@uwatwaterloo.ca  
University of Waterloo  
Canada

**ABSTRACT**

Ever since the birth of our field, RDBMSes and several classes of graph database management systems (GDBMSs) have existed side by side, providing a set of complementary features in data models, query languages, and visualization capabilities these data models provide. As a result, RDBMSs and GDBMSs appeal to different users for developing different sets of applications and there is immense value in extending RDBMSes to provide some capabilities of GDBMSs. We demonstrate GRainDB, a new system that extends advantages for extending RDBMSes to natively provide some of the capabilities of GDBMSs and support efficient graph querying. Over the past two years, we have started to develop a relational-core hybrid graph-relational system that we call GRainDB at the University of Waterloo. We use the term relational-core to indicate that GRainDB extends an RDBMS at its core. Specifically, GRainDB integrates a set of storage and query processing techniques, such as predefined pointer-based joins (reviewed in Section 4.1), into the columnar DocGraph RDBMS [2, 24] to make it more efficient on.
Add Path-finding

On top of the navigationally optimized joins, add path-finding algorithms

- **Bulk**: find cheapest paths between table of [src,dst] vertexes
- Bulk-optimizations: exploit landmarks, exploit SIMD

---

**SIGMOD’13**

Fast Exact Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling

Takuya Akiba  
The University of Tokyo, 113-0032, Japan  
t.akiba@is.s.u-tokyo.ac.jp

Yoichi Iwata  
The University of Tokyo, 113-0032, Japan  
y.iwata@is.s.u-tokyo.ac.jp

Yuichi Yoshida  
National Institute of Informatics,  
Preferred Infrastructure, Inc.  
Tokyo, 101-8430, Japan  
y.yoshida@nii.ac.jp

---

**BTW’17**

Efficient Batched Distance and Centrality Computation in Unweighted and Weighted Graphs

Manuel Then1, Stephan Günnemann2, Alfons Kemper3, Thomas Neumann4

Abstract: Distance and centrality computations are important building blocks for modern graph databases as well as for dedicated graph analytics systems. Two commonly used centrality metrics
Complexity of subgraph matching

Subgraph isomorphism is in NP but on graphs of bounded degree it is \textbf{polynomial}. Still, the complexity of evaluating a \textbf{triangle query with binary joins} is provably suboptimal, $O(|E|^2)$.

```
p1: Person  k  p2: Person
```

```
<table>
<thead>
<tr>
<th>i1</th>
<th>⋈</th>
<th>i2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t: Tag</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Triggered by many-to-many edges and skewed distributions.

Worst-case optimal join (\textbf{WCOJ}) algorithms are needed, which have a complexity of just $O(|E|^{1.5})$ for this query.

```
i1   ⋈ | i2   |
      |   |      |
      |   |      |
```

```
i1   ⋈ | k   |
      |   |      |
      |   |      |
```

```
i1   ⋈ | i2   |
      |   |      |
```

```
i1   ⋈ | k   |
      |   |      |
```

```
i1   ⋈ \ k
      |   |      |
```

```
i1 \ i2 \ k
      |   |      |
```

```
i1   \ i2 \ k
      |   |      |
```

Research on Worst-Case Optimal Joins (WCOJ)

Subject to research in the last ~15 years:

- **FOCS’08** bounds on complexity
- **PODS’12** Generic-Join (trie-based)
- **SIGMOD’16** GraphflowDB demo
- **PVLDB’19** query optimizer integration
- **PVLDB’20** hash-based WCOJ algorithm

Working implementations:

- **Industrial**: RelationalAI, LogicBlox, XTDB
- **Academic**: Umbra (umbra-db.com)
- **Open-source**: EdgeFrames (Spark, github.com/cwida/edge-frames)
Work on some of the missing pieces..

- Smart schema-discovering graph loading
- Property Graph Schema languages
- Vectorizable WCOJ algorithms
- Bulk “Cheapest Path” Finding Algorithms
- Relational Query Optimization that benefits graphs
- Transactional semantics for graph data
- ...

TPCTC’20

Towards Testing ACID Compliance in the LDBC Social Network Benchmark

Jack Uadly1, Benjamin A. Steere2, Karina Kustov3, Josef Marton4, Peter Bone3, and Gabor Salanyi5,6

1 Newcastle University, School of Computing, j.uadly@newcastle.ac.uk
2 Queen Mary University of London, b.a.steer@gmail.ac.uk
3 Budapest University of Technology and Economics
4 Department of Measurement and Information Systems
5 Budapest University of Technology and Economics
6 Department of Telecommunications and Media Informatics
7 CNT, Amsterdam, bszanyi@cs.uwaterloo.ca
8 MTA-SIME Lendület Cyber-Physical Systems Research Group

Abstract. Verifying ACID compliance is an essential part of database benchmarks; however, the intensity of performance analysis can be very...

M. Tamer Őzsu
University of Waterloo
David R. Cheriton School of Computer Science
https://cs.uwaterloo.ca/~tozu

TPCTC’20
SQL:2023 aka SQL/PGQ
SQL/PGQ: CREATE PROPERTY GRAPH

Major part of SQL:2023

Property Graph Definition (DDL) - Example

```sql
CREATE PROPERTY GRAPH aml

VERTEX TABLES ( account, customer )

LABEL customer PROPERTIES ( cid, name, city )

EDGE TABLES ( owns SOURCE customers DESTINATION accounts PROPERTIES ( since ) )

auth_signer SOURCE customer DESTINATION account

transfers SOURCE KEY ( from_id ) REFERENCES accounts ( aid )

DESTINATION KEY ( to_id ) REFERENCES accounts ( aid )

LABEL transfers PROPERTIES ( when, amount )
```

- Explicit label and properties options for customer
- Defaults apply for label and all properties.
- Columns when and amount are exposed as properties. Columns tid, from_id, and to_id are not.
SQL/PGQ: SELECT ... FROM GRAPH_TABLE

Major part of SQL:2023
(slides)

Querying PGs – Example 1

Access to ISO specs possible through liaison with LDBC. Become an LDBC member!
Graph Query Language (GQL)

New ISO standard with Cypher-like syntax:

```sql
USE my_social_graph
MATCH (p:Person)-[:FRIEND*{1,2}]->(friend_or_foaf)
WHERE friend_or_foaf.age > $age AND p.country = $country
RETURN count(*)
```

Will also support returning graphs. Unsure timeline.

https://gqlstandards.org
https://ldbcouncil.org/event/fourteenth-tuc-meeting/attachments/stefan-plantikow-gql.pdf
Conclusions
Conclusion

- Discussed the relationship between GDBMS and RDBMS
- Graph queries have interesting use cases, and their usage will continue to expand
- LDBC has created useful benchmarks, but also query and schema languages
  - LDBC Technical User Community Meeting at SIGMOD’22 on Friday
- Current generation of GDBMS is often not competent
- Discussed pitfalls (“6 blunders”) in GDBMS architectures
- Outlined future standards SQL/PGQ in SQL:2023 (and.. GQL)
- Outlined the blueprint of a competent GDBMS
  - CWI is building a PGQ extension module for DuckDB

Gábor Szárnyas

Hannes Mühleisen & Mark Raasveldt