CWI
Centrum Wiskunde & Informatica

VU

The (sorry) State of Graph Database Systems

Peter Boncz
CWI

comparing graph with relational database systems..

+ *provide pointers to related literature*

CWI
Centrum Wiskunde & Informatica

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)

LDBC
The graph & RDF benchmark reference

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

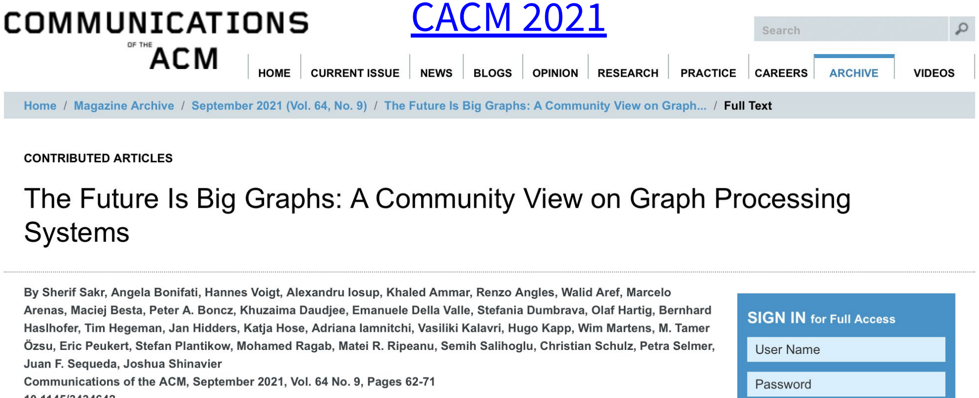
Gained a foothold in the data systems market

- 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**)

CACM 2021



COMMUNICATIONS OF THE ACM
Search
HOME
CURRENT ISSUE
NEWS
BLOGS
OPINION
RESEARCH
PRACTICE
CAREERS
ARCHIVE
VIDEOS
Home / Magazine Archive / September 2021 (Vol. 64, No. 9) / The Future Is Big Graphs: A Community View on Graph... / Full Text
SIGN IN for Full Access
User Name
Password

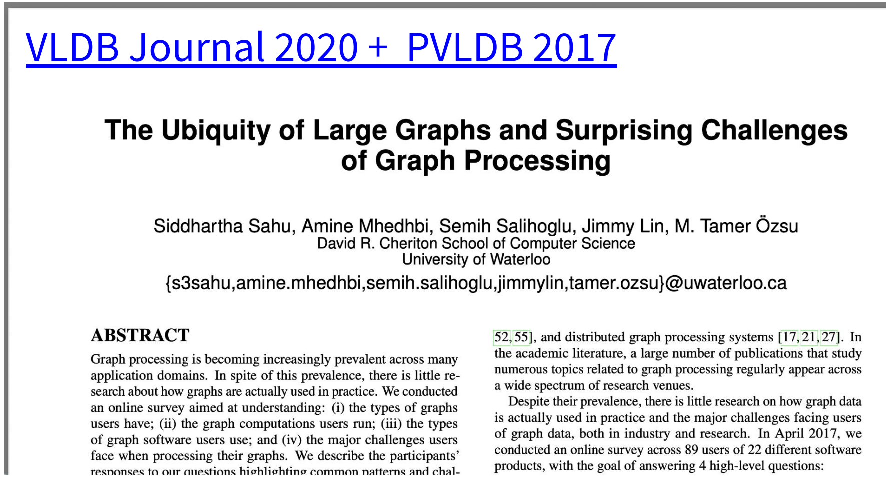
CONTRIBUTED ARTICLES

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

By Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei R. Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier

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

VLDB Journal 2020 + PVLDB 2017



The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing

Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, M. Tamer Özsu
David R. Cheriton School of Computer Science
University of Waterloo
{s3sahu,amine.mhedhbi,semih.salihoglu,jimmylin,tamer.ozsu}@uwaterloo.ca

ABSTRACT

Graph processing is becoming increasingly prevalent across many application domains. In spite of this prevalence, there is little research about how graphs are actually used in practice. We conducted an online survey aimed at understanding: (i) the types of graphs users have; (ii) the graph computations users run; (iii) the types of graph software users use; and (iv) the major challenges users face when processing their graphs. We describe the participants' responses to our questions highlighting common patterns and chal-

52, 55], and distributed graph processing systems [17, 21, 27]. In the academic literature, a large number of publications that study numerous topics related to graph processing regularly appear across a wide spectrum of research venues.

Despite their prevalence, there is little research on how graph data is actually used in practice and the major challenges facing users of graph data, both in industry and research. In April 2017, we conducted an online survey across 89 users of 22 different software products, with the goal of answering 4 high-level questions:

Key GDBMS building blocks

⚫->⚫
property graph data model

🌀
graph query language

🔎
graph visualization

🌀
subgraph matching

🌀
relational queries

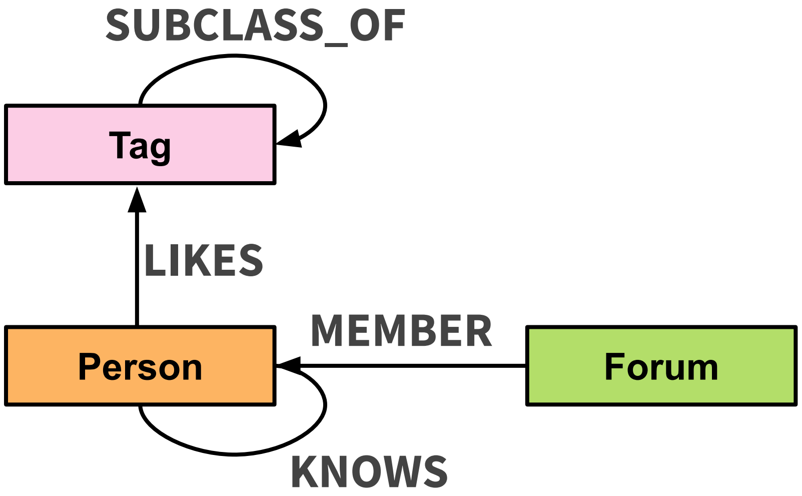
🌀
path queries

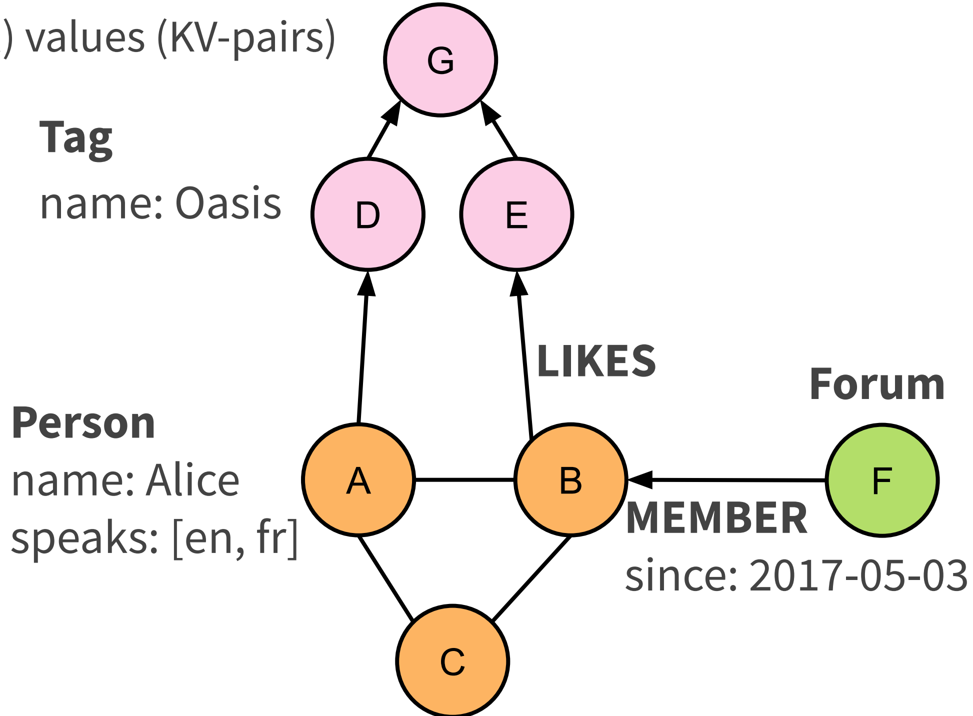
🌀
stored procedures

Data model: Property graph

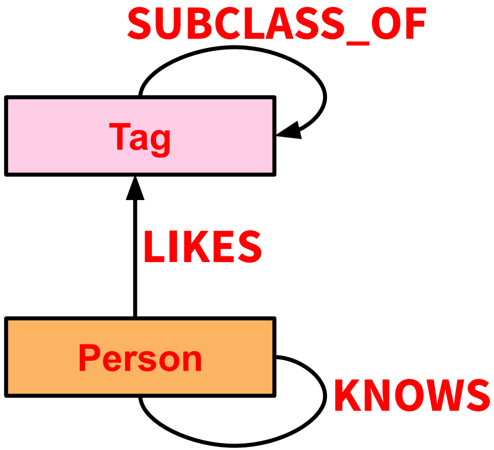
Directed graph consisting of **labeled** entities: vertexes & edges

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



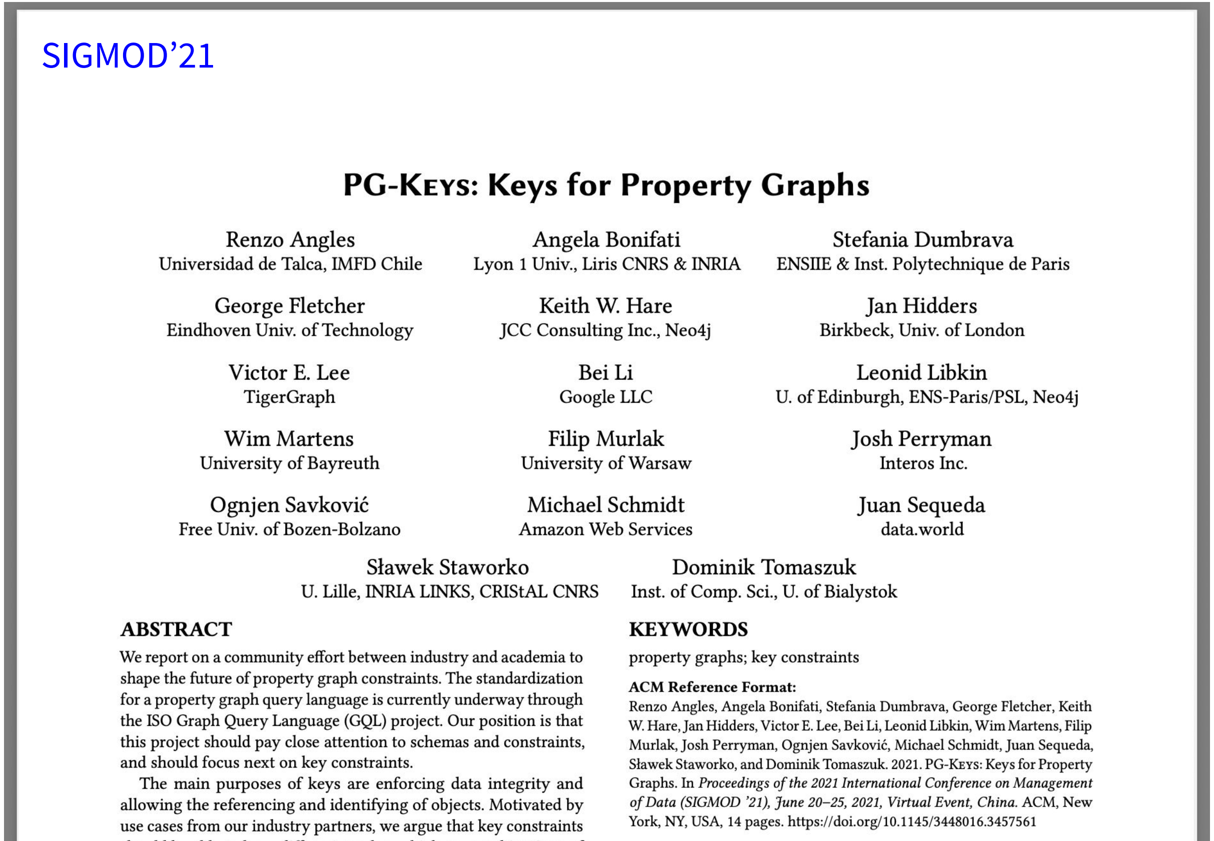


LDBC Property Graph Schema Working Group



Topics of study:

- Constraints
- Properties
- Nulls



SIGMOD'21

PG-Keys: Keys for Property Graphs

Renzo Angles
Universidad de Talca, IMFD Chile

Angela Bonifati
Lyon 1 Univ., Liris CNRS & INRIA

Stefania Dumbrava
ENSIIE & 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

Bei 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 Savković
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 Bialystok

ABSTRACT

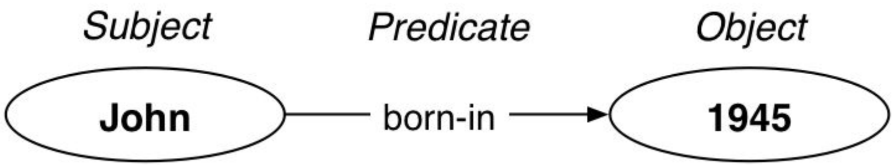
We report on a community effort between industry and academia to shape the future of property graph constraints. The standardization for a property graph query language is currently underway through the ISO Graph Query Language (GQL) project. Our position is that this project should pay close attention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

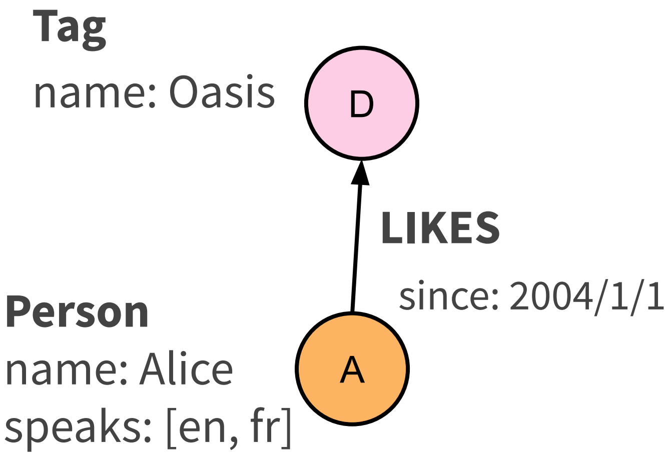
KEYWORDS

property graphs; key constraints

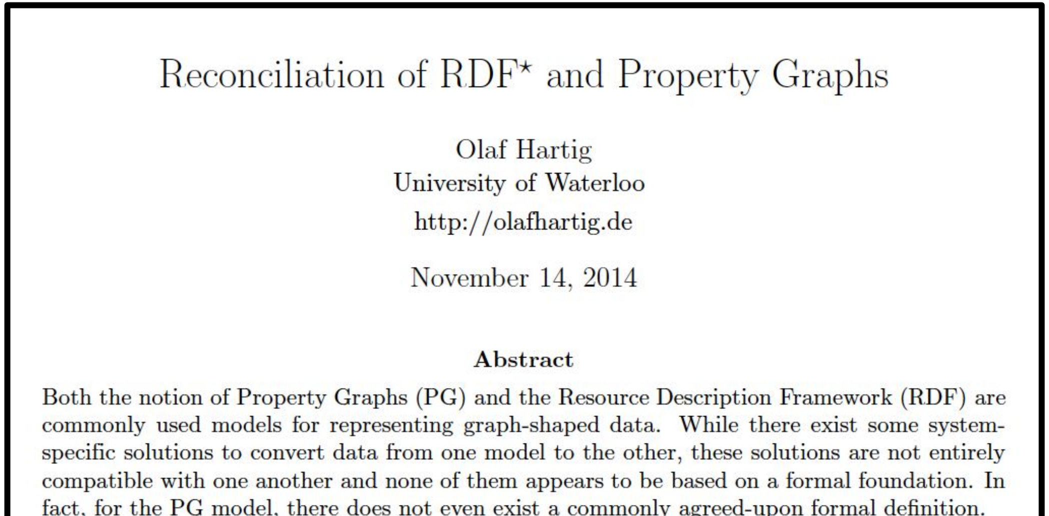
ACM Reference Format:
Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savković, Michael Schmidt, Juan Sequeda, Sławek Staworko, and Dominik Tomaszuk. 2021. PG-Keys: Keys for Property Graphs. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457561



Data model: RDF triples



vs Property Graph



Reconciliation of RDF* and Property Graphs

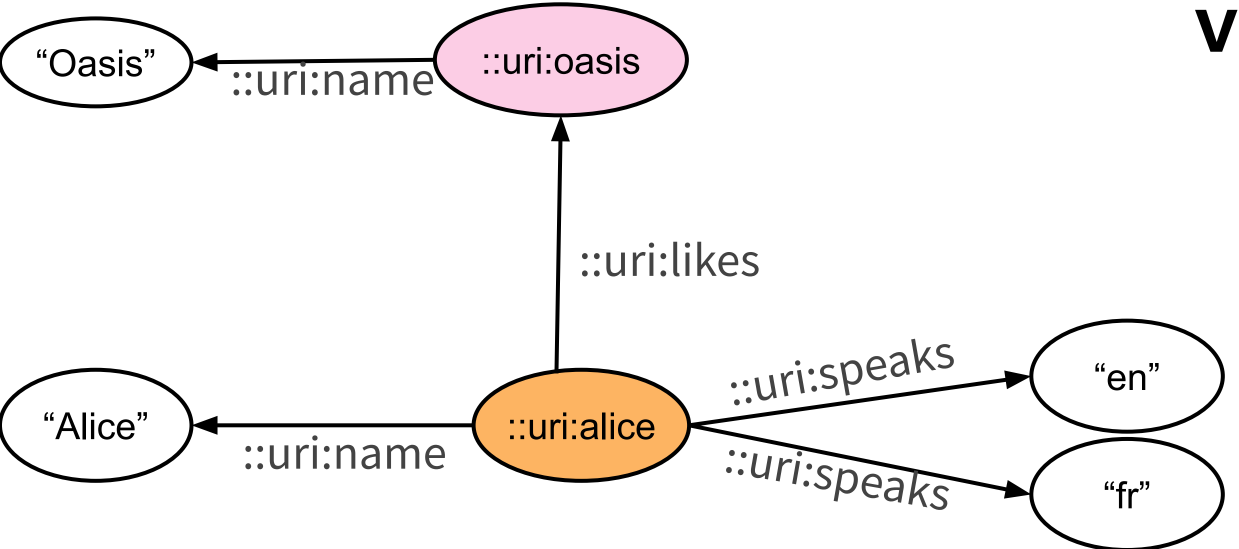
Olaf Hartig
University of Waterloo
http://olafhartig.de

November 14, 2014

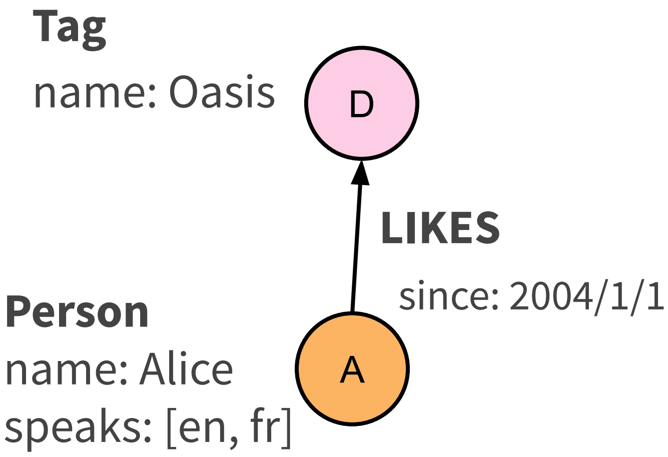
Abstract

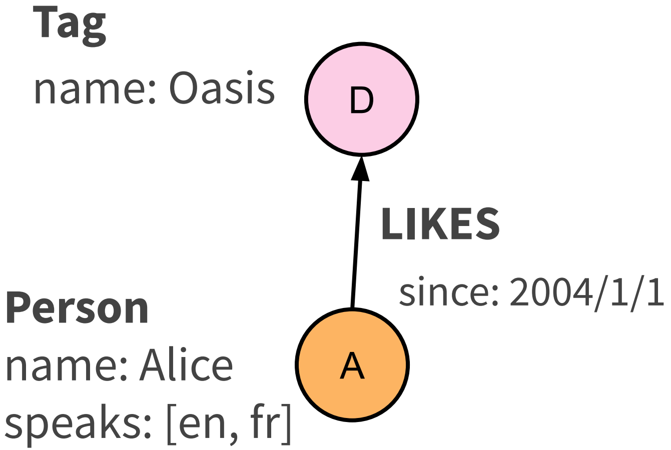
Both the notion of Property Graphs (PG) and the Resource Description Framework (RDF) are commonly used models for representing graph-shaped data. While there exist some system-specific solutions to convert data from one model to the other, these solutions are not entirely compatible with one another 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

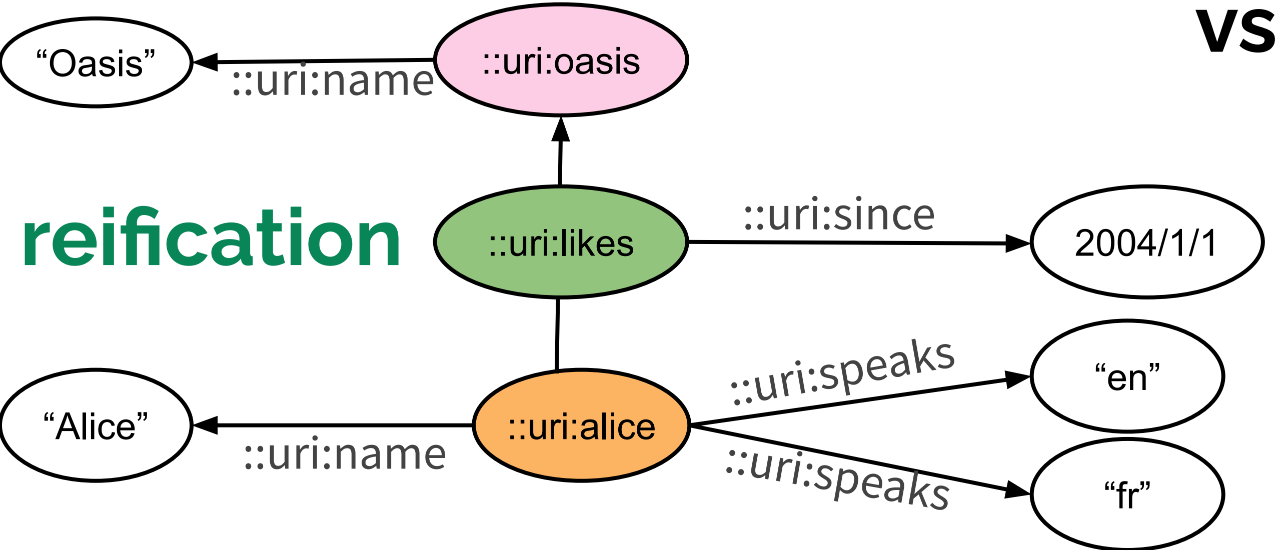




Data model: RDF triples

vs Property Graph

reification



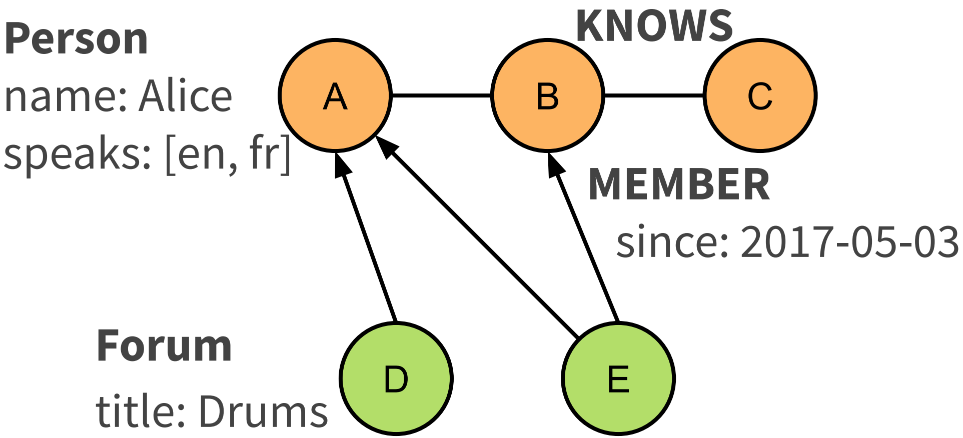
Query language: Cypher

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

subgraph matching

path query (Kleene-star)

relational operators



Isomorphic semantics

```
|"p1"|"f"|"count(p2)"|"count(p3)"|
|{B} |{E}|1          |1          |
```

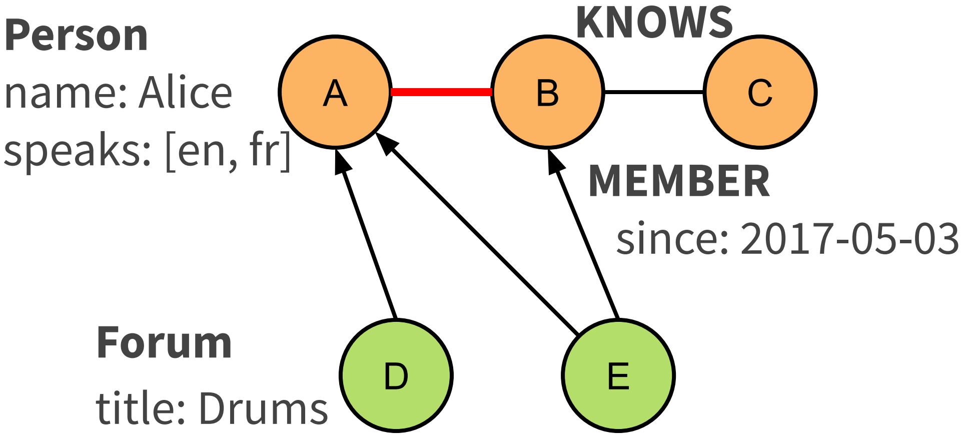
Query language: Cypher

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

subgraph matching

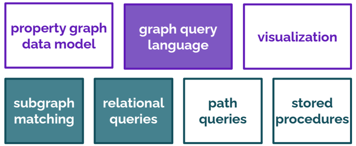
path query (Kleene-star)

relational operators



Homomorphic semantics

```
╒════╤═════╤═══════════╤═══════════╕
│"p1"│"f"  │"count(p2)"│"count(p3)"│
╞════╪═════╪═══════════╪═══════════╡
│{B} │{E}  │1          │1          │
│{A} │{E}  │1          │1          │
└────┴─────┴───────────┴───────────┘
```



property graph data model

graph query language

visualization

subgraph matching

relational queries

path queries

stored procedures

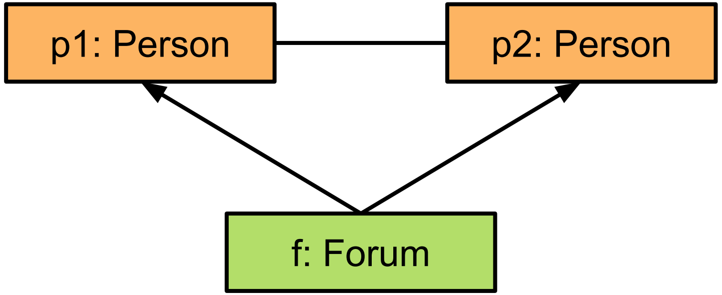
Pattern matching

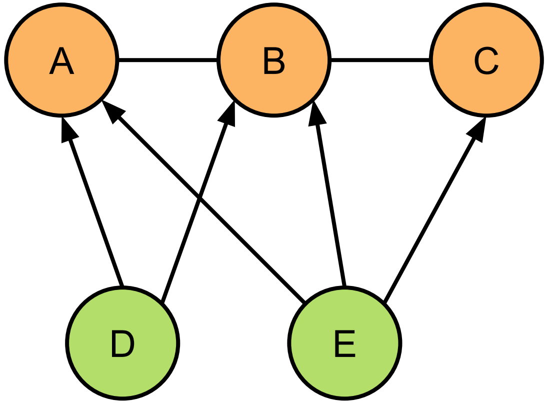
- **basic graph pattern**
- complex graph pattern

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)

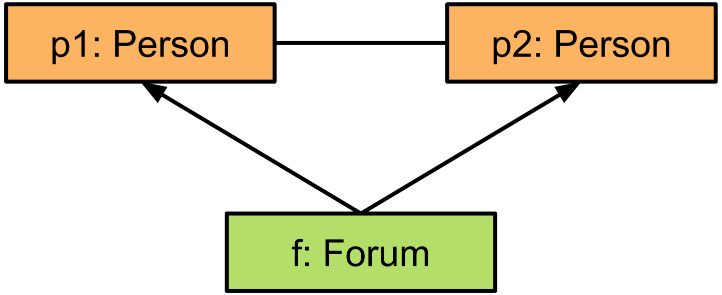
(A, B, E)

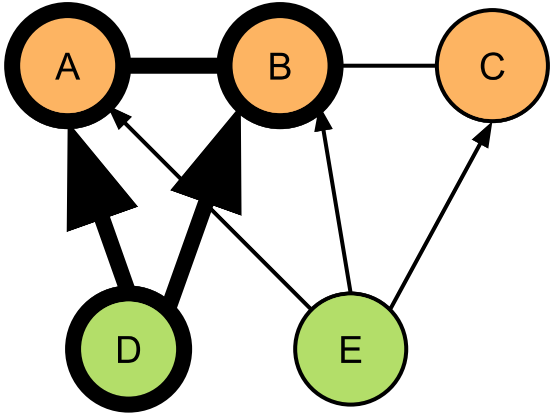
(B, C, E)

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)

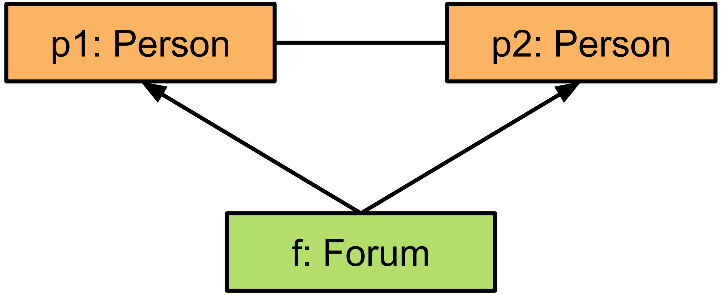
(B, C, E)

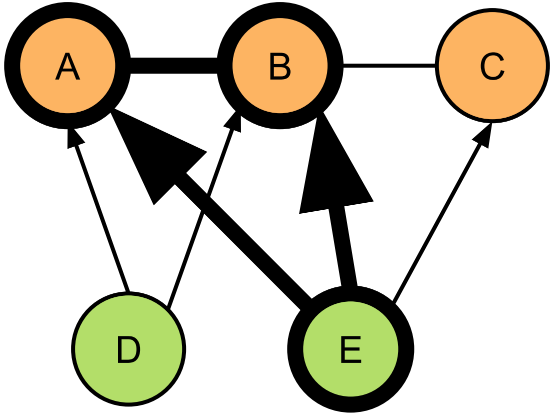
CWI
Centrum Wiskunde & Informatica

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)

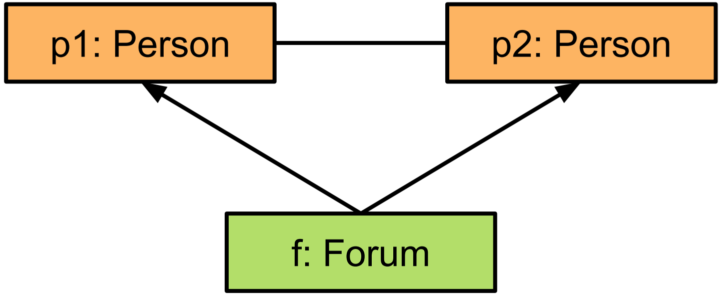
(A, B, E)

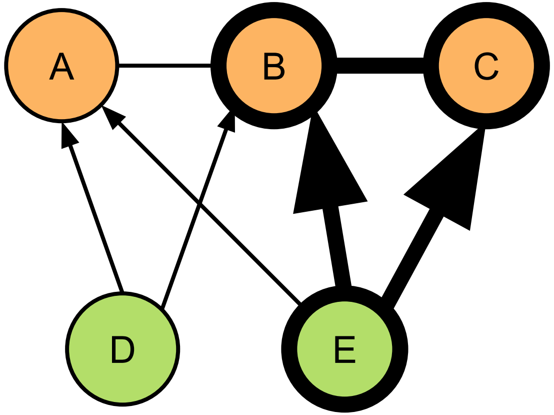
(B, C, E)

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)

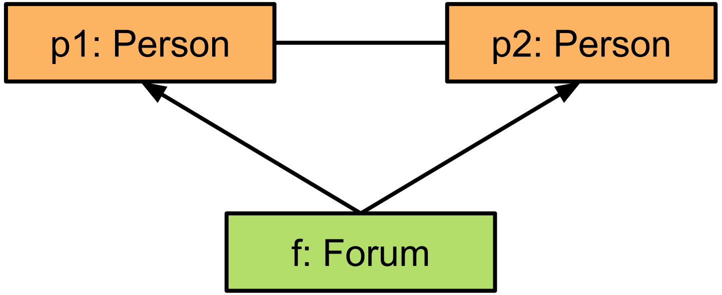
(A, B, E)

(B, C, E)

Subgraph matching (SQL)

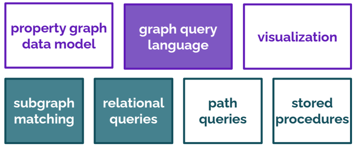
Edge tables: `knows(person1id, person2id); member(forumid, personid)`

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
```

Q: m1 ⋈ m2 ⋈ knows



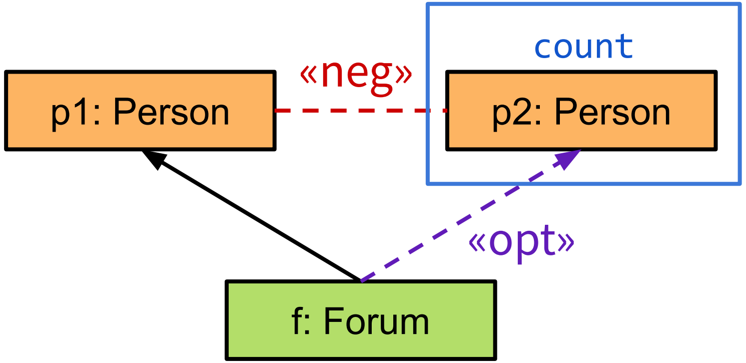
Pattern matching

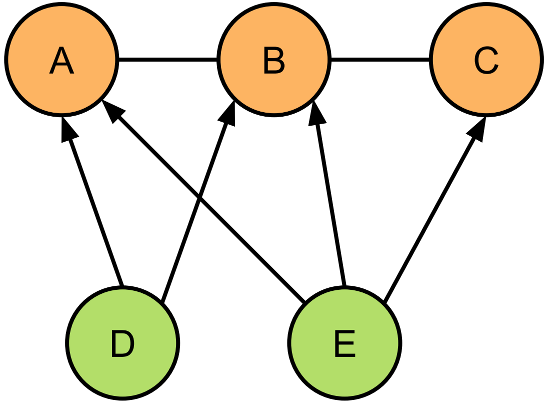
- basic graph pattern
- **complex graph pattern**

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:

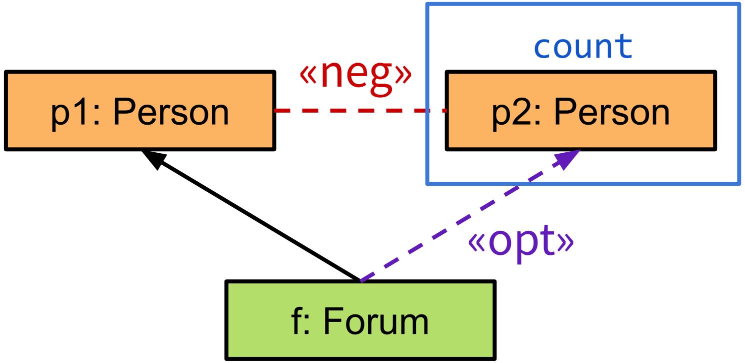
(D, 0)

(E, 1)

Subgraph matching (SQL)

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

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

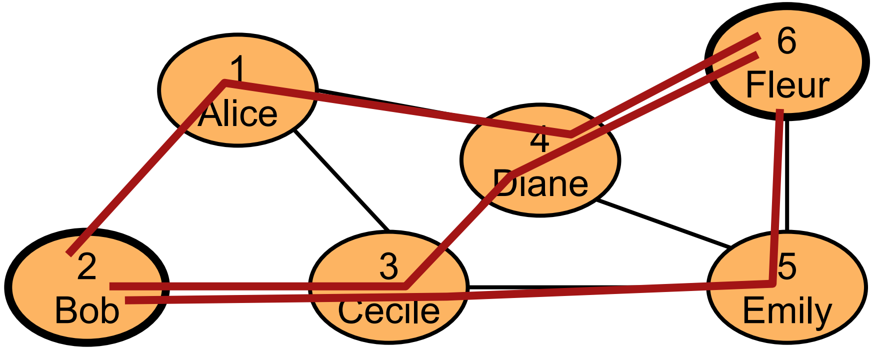


```
SELECT m1.forumid, count(m2.personid)
FROM member m1
LEFT OUTER JOIN member m2
             ON m1.forumid = m2.forumid
            AND m1.personid < m2.personid
WHERE NOT EXISTS (SELECT true FROM knows
  WHERE person1id = m1.personid
    AND person2id = m2.personid)
GROUP BY m1.forumid
```

$\gamma\,(\,\text{member} ⟕ \text{member} \triangleright \text{knows}\,)$

Unweighted shortest path in Cypher

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

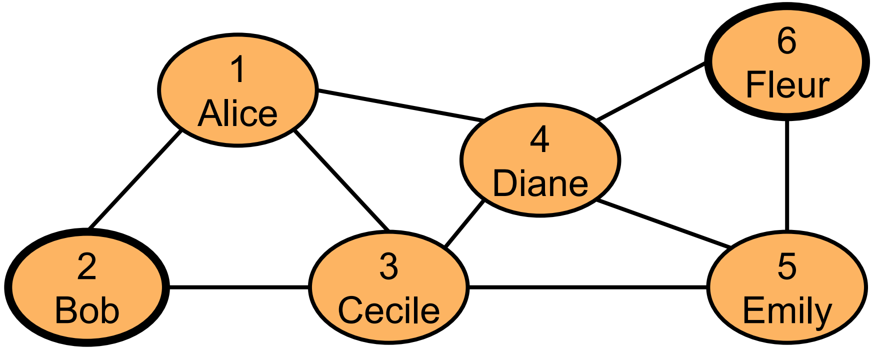


Result:

```
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

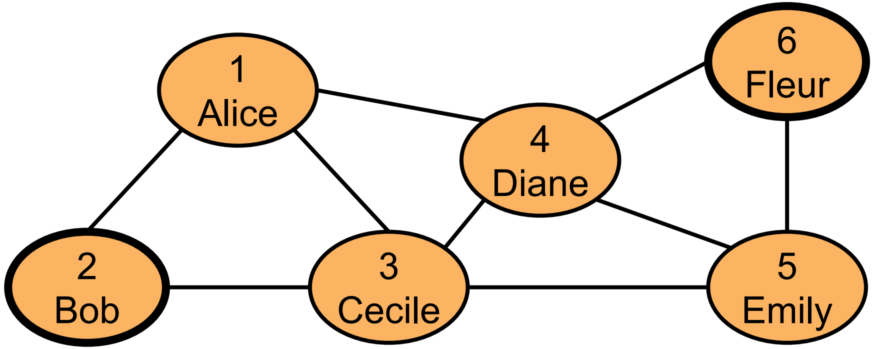
id [PK]	name
1	Alice
2	Bob
3	Cecile
4	Diane
5	Emily
6	Fleur

knows

person1id [FK]	person2id [FK]
1	2
1	3
1	4
2	3
3	4
3	5
4	5
4	6
5	6
all edges backwards	

Unw. SP query: Data in SQL

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



Person

	id [PK]	name
	1	Alice
source →	2	Bob
	3	Cecile
	4	Diane
	5	Emily
target →	6	Fleur

knows

person1id [FK]	person2id [FK]
1	2
1	3
1	4
2	3
3	4
3	5
4	5
4	6
5	6
all edges backwards	

Unweighted shortest path query in SQL

```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;
```

```
┌───────────────────┬─────────┬───────────────┐
│       path        │  level  │ targetReached │
├───────────────────┼─────────┼───────────────┤
│ [2, 3]            │ 1       │ false         │
│ [2, 1]            │ 1       │ false         │
│ [2, 1, 4]         │ 2       │ false         │
│ [2, 3, 5]         │ 2       │ false         │
│ [2, 3, 4]         │ 2       │ false         │
│ [2, 1, 4, 6]      │ 3       │ true          │
│ [2, 3, 5, 6]      │ 3       │ true          │
│ [2, 3, 4, 6]      │ 3       │ true          │
└───────────────────┴─────────┴───────────────┘
```

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;
```

initial edge

source person

"reached target node?" using a window function

add next edge

cycle detection in path

not reached target node

prevent visiting nodes already found in previous steps

```
┌───────────────────┬─────────┬───────────────┐
│       path        │  level  │ targetReached │
├───────────────────┼─────────┼───────────────┤
│ [2, 3]            │ 1       │ false         │
│ [2, 1]            │ 1       │ false         │
│ [2, 1, 4]         │ 2       │ false         │
│ [2, 3, 5]         │ 2       │ false         │
│ [2, 3, 4]         │ 2       │ false         │
│ [2, 1, 4, 6]      │ 3       │ true          │
│ [2, 3, 5, 6]      │ 3       │ true          │
│ [2, 3, 4, 6]      │ 3       │ true          │
└───────────────────┴─────────┴───────────────┘
```

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))
)
```

```
┌──────────────────┬────────┬───────────────┐
│       path       │ level  │ targetReached │
├──────────────────┼────────┼───────────────┤
│ [2, 3]           │ 1      │ false         │
│ [2, 1]           │ 1      │ false         │
│ [2, 1, 4]        │ 2      │ false         │
│ [2, 3, 5]        │ 2      │ false         │
│ [2, 3, 4]        │ 2      │ false         │
│ [2, 1, 4, 6]     │ 3      │ true          │
│ [2, 3, 5, 6]     │ 3      │ true          │
│ [2, 3, 4, 6]     │ 3      │ true          │
└──────────────────┴────────┴───────────────┘
```

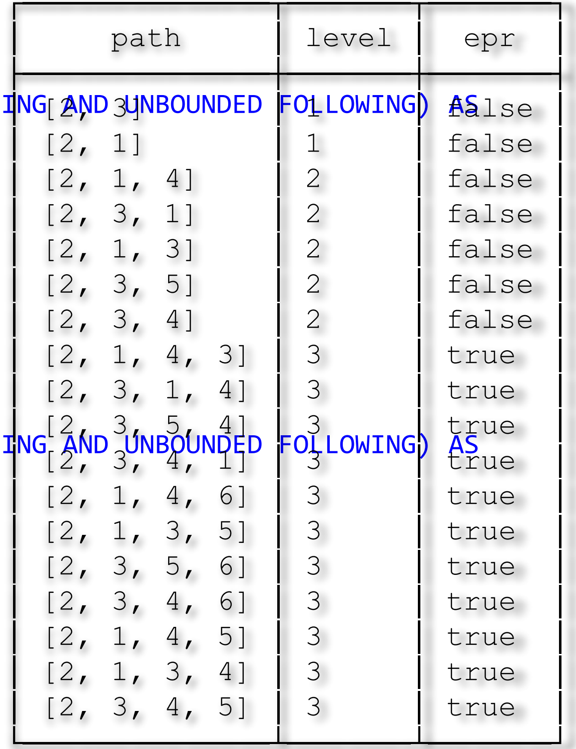
finding the names on the paths by unnesting & joining

```
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;
```

```
┌────────────────────────────────┐
│           pathNames            │
├────────────────────────────────┤
│ [Bob, Alice, Diane, Fleur]     │
│ [Bob, Cecile, Emily, Fleur]    │
│ [Bob, Cecile, Diane, Fleur]    │
└────────────────────────────────┘
```

Unweighted shortest path query in SQL

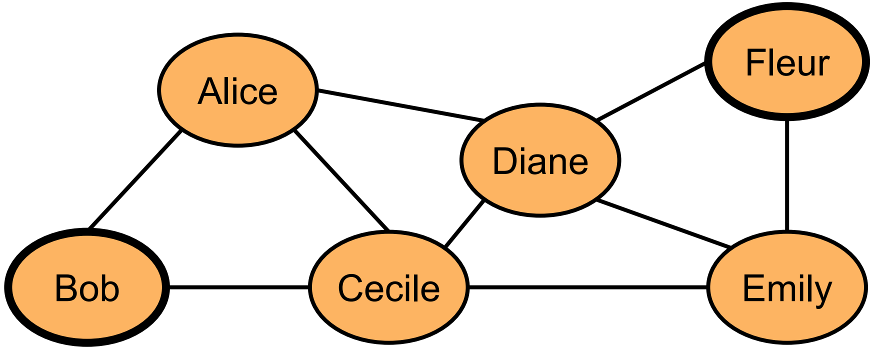
```
WITH RECURSIVE paths(startPerson, endPerson, path, level, endPersonReached) AS (
   SELECT person1id AS startPerson, person2id AS endPerson,
          [person1id, person2id]::bigint[] AS path, 1 AS level,
          max(CASE WHEN p2.name = 'Fleur'
            THEN true ELSE false END) OVER (ROWS BETWEEN 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;
```



path	level	epr
[2, 3]	1	false
[2, 1]	1	false
[2, 1, 4]	2	false
[2, 3, 1]	2	false
[2, 1, 3]	2	false
[2, 3, 5]	2	false
[2, 3, 4]	2	false
[2, 1, 4, 3]	3	true
[2, 3, 1, 4]	3	true
[2, 3, 5, 4]	3	true
[2, 3, 4, 1]	3	true
[2, 1, 4, 6]	3	true
[2, 1, 3, 5]	3	true
[2, 3, 5, 6]	3	true
[2, 3, 4, 6]	3	true
[2, 1, 4, 5]	3	true
[2, 1, 3, 4]	3	true
[2, 3, 4, 5]	3	true

Unweighted shortest path in Cypher

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

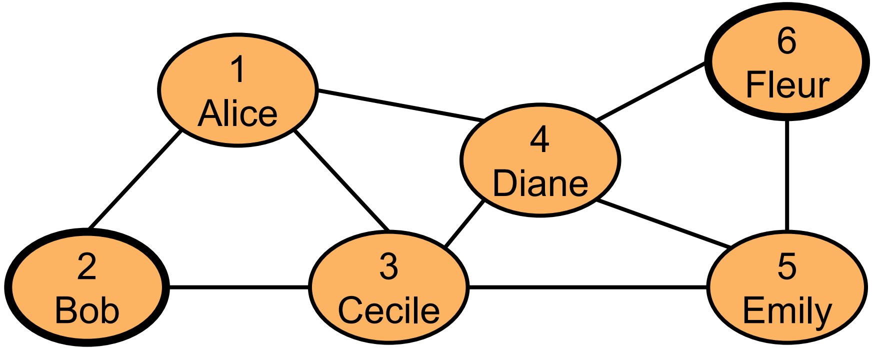


Result:

```
length
------
     3
```

Unw. SP query: Data in SQL

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



Person

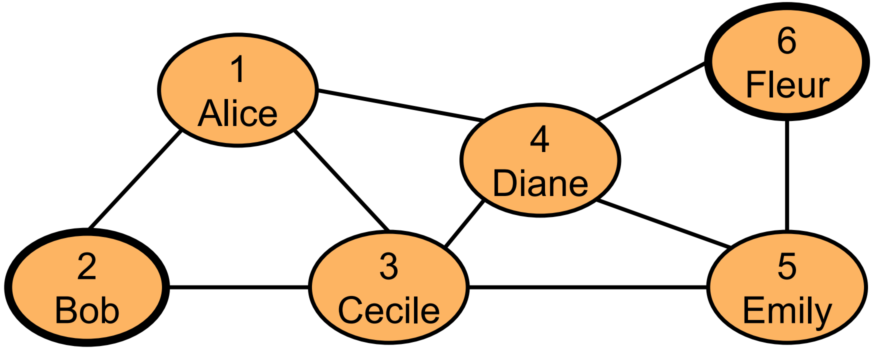
id [PK]	name
1	Alice
2	Bob
3	Cecile
4	Diane
5	Emily
6	Fleur

knows

person1id [FK]	person2id [FK]
1	2
1	3
1	4
2	3
3	4
3	5
4	5
4	6
5	6
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)



Person

	id [PK]	name
	1	Alice
start	2	Bob
	3	Cecile
	4	Diane
	5	Emily
end	6	Fleur

knows

person1id [FK]	person2id [FK]
1	2
1	3
1	4
2	3
3	4
3	5
4	5
4	6
5	6
all edges backwards (optional)	

Unweighted shortest path query in SQL

```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;
```

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 PR
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 PRECE
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;
```

initial edge

reached end node? w/ window function

adding an edge to the path

reached end node? w/ window function

cycle detection

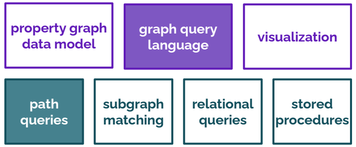
check reached end node

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

path	level
[2, 1, 4, 6]	3
[2, 3, 5, 6]	3
[2, 3, 4, 6]	3



property graph data model

graph query language

visualization

path queries

subgraph matching

relational queries

stored procedures

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

neo4j

Cypher

Oracle Labs
PGX

PGQL

TigerGraph

GSQL

XTDB

Datalog

relationalAI

Rel

Dgraph

DQL

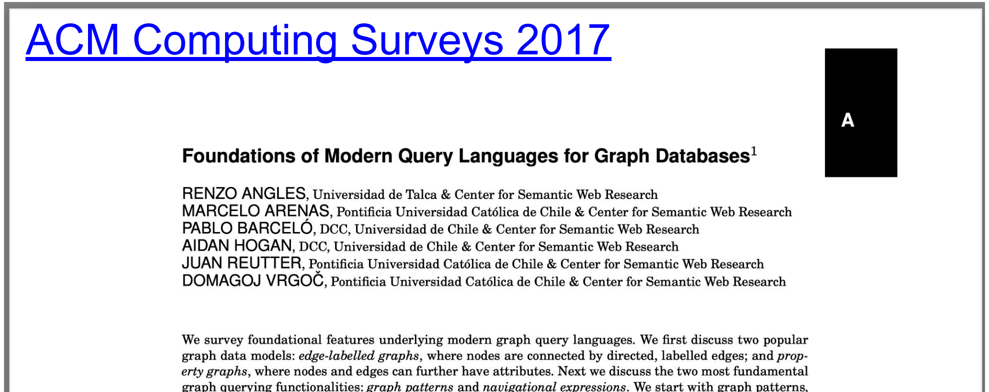
Amazon Neptune

SPARQL, Cypher, Gremlin

JanusGraph

Gremlin

See also:
A Survey of Current Property Graph Query Languages
(2021) by Peter Boncz &&



ACM Computing Surveys 2017

A

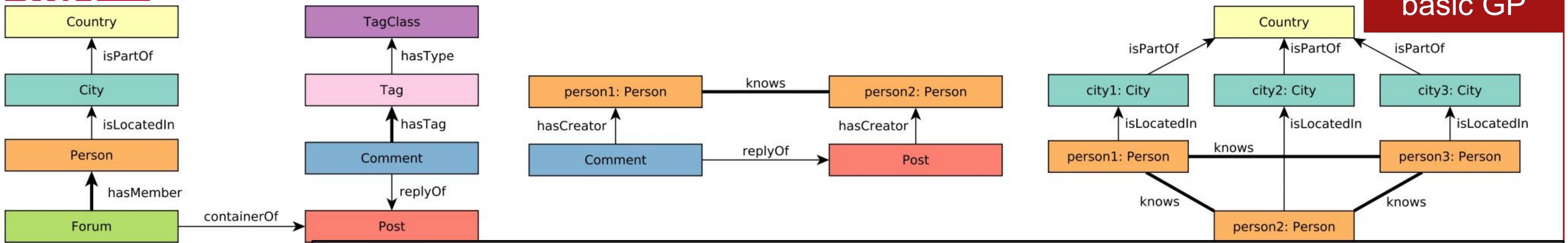
Foundations of Modern Query Languages for Graph Databases[1]

RENZO ANGLES, Universidad de Talca & Center for Semantic Web Research
MARCELO ARENAS, Pontificia Universidad Católica de Chile & Center for Semantic Web Research
PABLO BARCELÓ, DCC, Universidad de Chile & Center for Semantic Web Research
AIDAN HOGAN, DCC, Universidad de Chile & Center for Semantic Web Research
JUAN REUTTER, Pontificia Universidad Católica de Chile & Center for Semantic Web Research
DOMAGOJ VRGOČ, Pontificia Universidad Católica de Chile & Center for Semantic Web Research

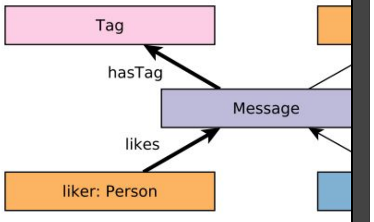
We survey foundational features underlying modern graph query languages. We first discuss two popular graph data models: *edge-labelled graphs*, where nodes are connected by directed, labelled edges; and *property graphs*, where nodes and edges can further have attributes. Next we discuss the two most fundamental graph querying functionalities: *graph patterns* and *navigational expressions*. We start with graph patterns,

A simple test of Graph Data Systems

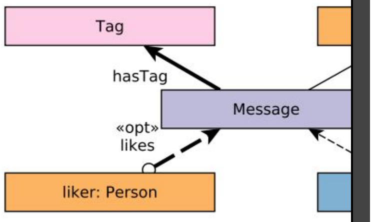
basic GP



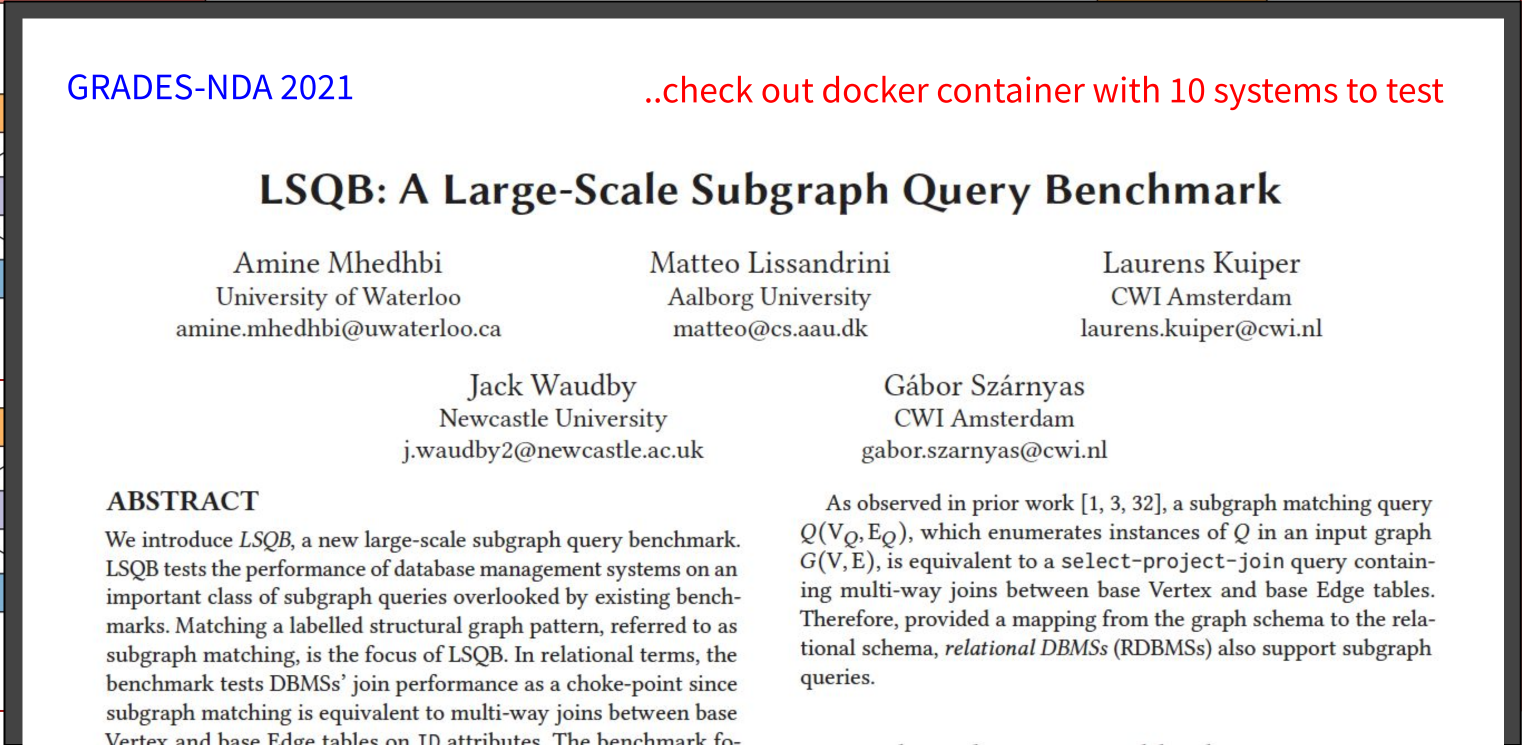
(a) Q1.



(d) Q4.



(g) Q7.



GRADES-NDA 2021

..check out docker container with 10 systems to test

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

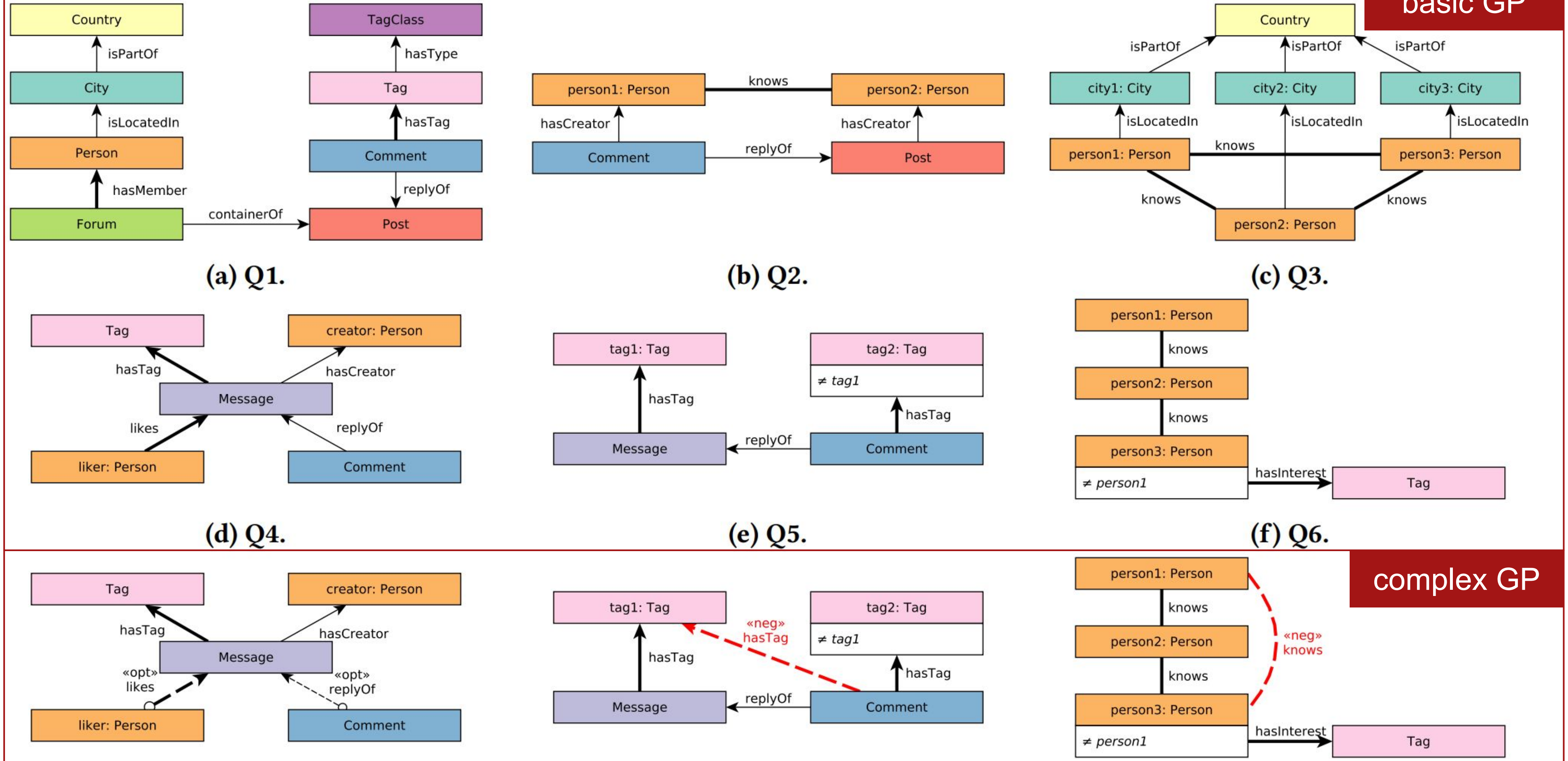
Gábor Szárnyas
CWI Amsterdam
gabor.szarnyas@cwi.nl

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 fo-

As observed in prior work [1, 3, 32], a subgraph matching query $Q(V_Q, E_Q)$, which enumerates instances of Q 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, *relational DBMSs* (RDBMSs) also support subgraph queries.

basic GP



(a) Q1.

(b) Q2.

(c) Q3.

(d) Q4.

(e) Q5.

(f) Q6.

complex GP

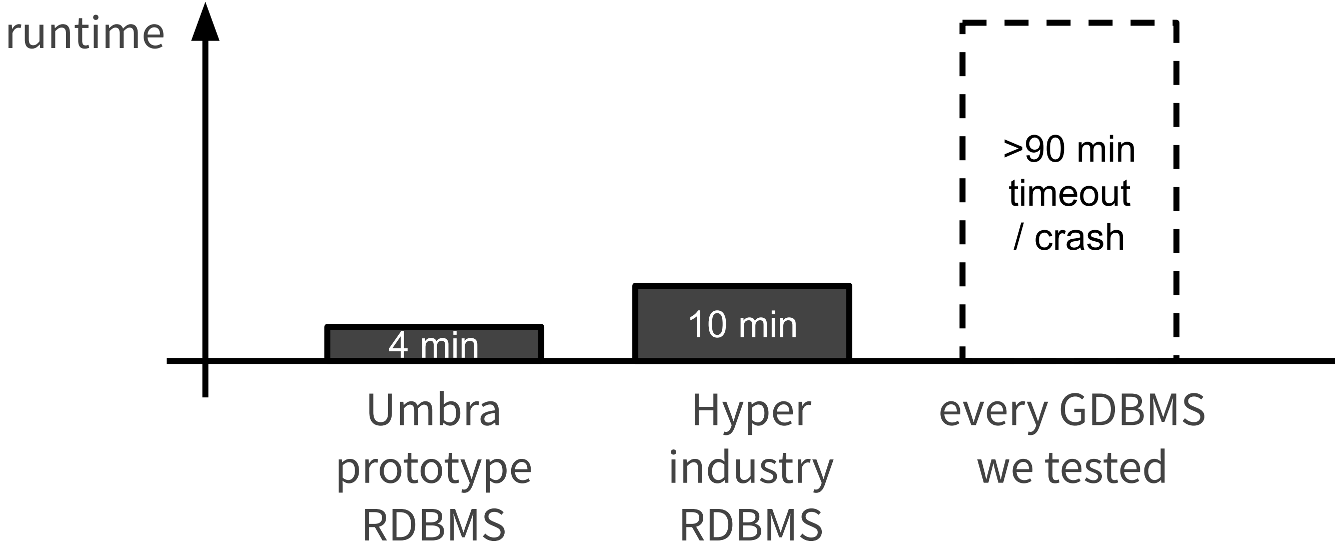
(g) Q7.

(h) Q8.

(i) Q9.

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 datasize 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

book query:

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

Throwing all edges in one basket: a good idea?

O	P	S
0		
1995	year	4
1996	year	0
1996	year	6
1997	year	3
1998	year	2
ZZZZ		

⋈

P	S	O
a		
isbn	0	i1996
isbn	2	i1998
isbn	3	i1997
isbn	4	i1995
isbn	6	i1996
z		

⋈

P	S	O
a		
auth	0	a1996
auth	2	a1998
auth	3	a1997
auth	4	a1995
auth	5	foo
auth	6	a1996

an indexing on all 6 triple orders does not guarantee access locality (**red**)!!

- relational **clustered index**

year	author	isbn
1975	a1995	i1995
1996	a1996	i1996
1996	a1996	i1996
1997	a1997	i1997
1998	a1998	i1998

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

- relational **partitioned table**

1995

author	isbn
a1995	i1995

1996

author	isbn
a1996	i1996
a1996	i1996

1997

author	isbn
a1997	i1997

1998

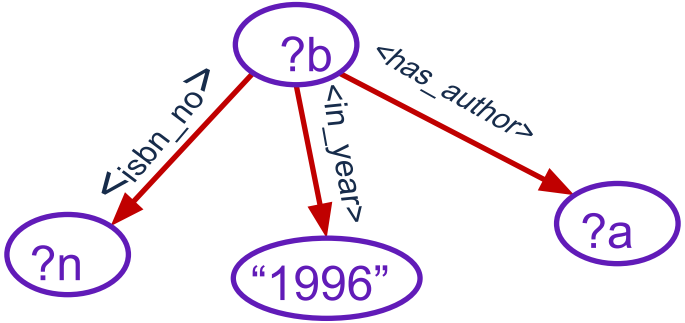
author	isbn
a1998	i1998

Triple Fallacy 2: Join Jungle

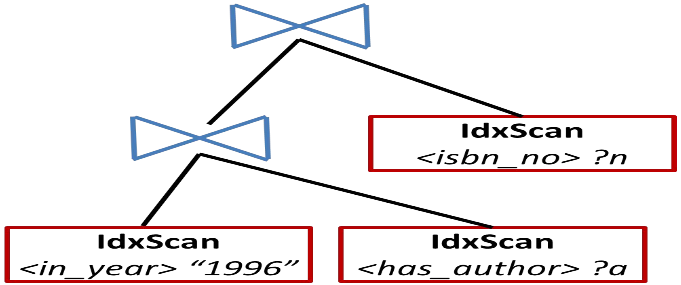
- superfluous joins **explode** query complexity

book query:

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



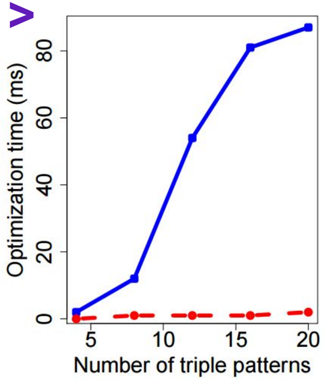
< Query graph>



< Example query plan >

- 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



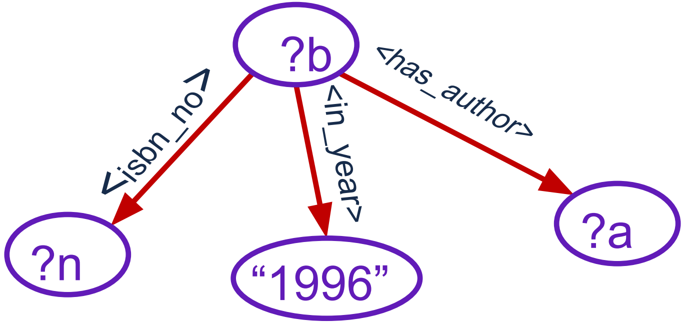
(b) Virtuoso

Triple Fallacy 3: Cardinality Crisis

book query:

```
SELECT ?a ?n WHERE {
  ?b <has_author> ?a.
  ?b <in_year>   “1996”.
  ?b <isbn_no>    ?n
}
```

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



< Query graph>

- because of **structural correlations**
  - if (?b has an <isbn_no>) it’s a book, it has <in_year> and <has_author>
  - query optimizer estimates using the **independence assumption**
  - many joins (fallacy 2) + wrong estimates ⇒ 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

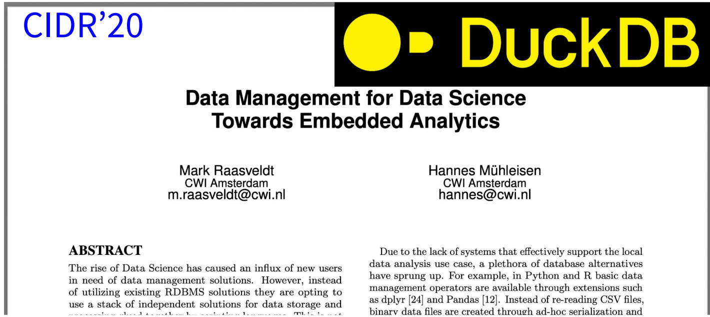
⇒ Query languages should only allow tractable queries, e.g.

- 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



CIDR'20

DuckDB

Data Management for Data Science Towards Embedded Analytics

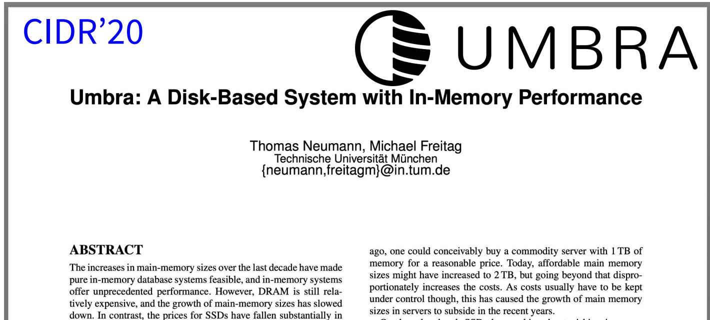
Mark Raasveldt
CWI Amsterdam
m.raasveldt@cwi.nl

Hannes Mühleisen
CWI Amsterdam
hannes@cwi.nl

ABSTRACT

The rise of Data Science has caused an influx of new users in need of data management solutions. However, instead of utilizing existing RDBMS solutions they are opting to use a stack of independent solutions for data storage and

Due to the lack of systems that effectively support the local data analysis use case, a plethora of database alternatives have sprung up. For example, in Python and R basic data management operators are available through extensions such as dplyr [24] and Pandas [12]. Instead of re-reading CSV files, binary data files are created through ad-hoc serialization and



CIDR'20

UMBRA

Umbra: A Disk-Based System with In-Memory Performance

Thomas Neumann, Michael Freitag
Technische Universität München
{neumann,freitagm}@in.tum.de

ABSTRACT

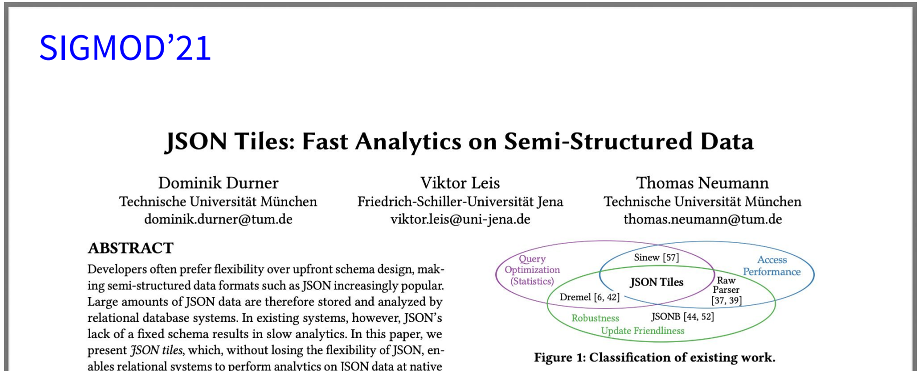
The increases in main-memory sizes over the last decade have made pure in-memory database systems feasible, and in-memory systems offer unprecedented performance. However, DRAM is still relatively expensive, and the growth of main-memory sizes has slowed down. In contrast, the prices for SSDs have fallen substantially in

ago, one could conceivably buy a commodity server with 1 TB of memory for a reasonable price. Today, affordable main memory sizes might have increased to 2 TB, but going beyond that disproportionately increases the costs. As costs usually have to be kept under control though, this has caused the growth of main memory sizes in servers to subside in the recent years.

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



SIGMOD’21

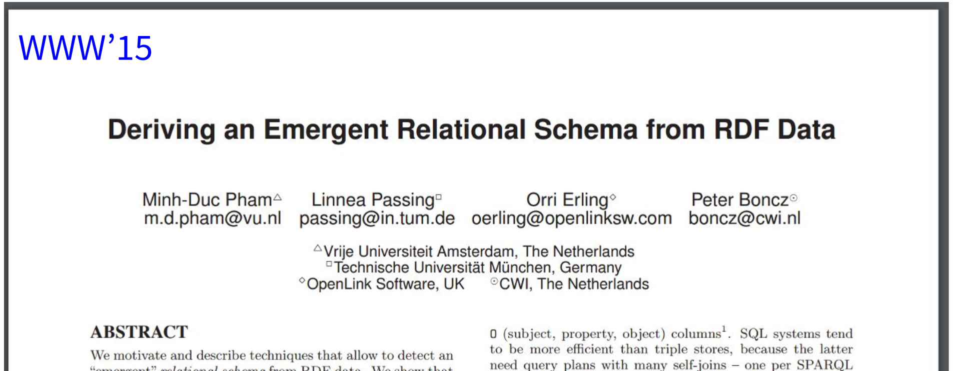
JSON Tiles: Fast Analytics on Semi-Structured Data

Dominik Durner, Technische Universität München, dominik.durner@tum.de
Viktor Leis, Friedrich-Schiller-Universität Jena, viktor.leis@uni-jena.de
Thomas Neumann, Technische Universität München, thomas.neumann@tum.de

ABSTRACT

Developers often prefer flexibility over upfront schema design, making semi-structured data formats such as JSON increasingly popular. Large amounts of JSON data are therefore stored and analyzed by relational database systems. In existing systems, however, JSON’s lack of a fixed schema results in slow analytics. In this paper, we present *JSON tiles*, which, without losing the flexibility of JSON, enables relational systems to perform analytics on JSON data at native

Figure 1: Classification of existing work.



WWW’15

Deriving an Emergent Relational Schema from RDF Data

Minh-Duc Pham[△] m.d.pham@vu.nl
Linnea Passing[□] passing@in.tum.de
Orri Erling[◇] oerling@openlinksw.com
Peter Boncz[⊙] boncz@cwi.nl

[△] Vrije Universiteit Amsterdam, The Netherlands
[□] Technische Universität München, Germany
[◇] OpenLink Software, UK [⊙] CWI, The Netherlands

ABSTRACT

We motivate and describe techniques that allow to detect an “emergent” *relational schema* from RDF data. We show that

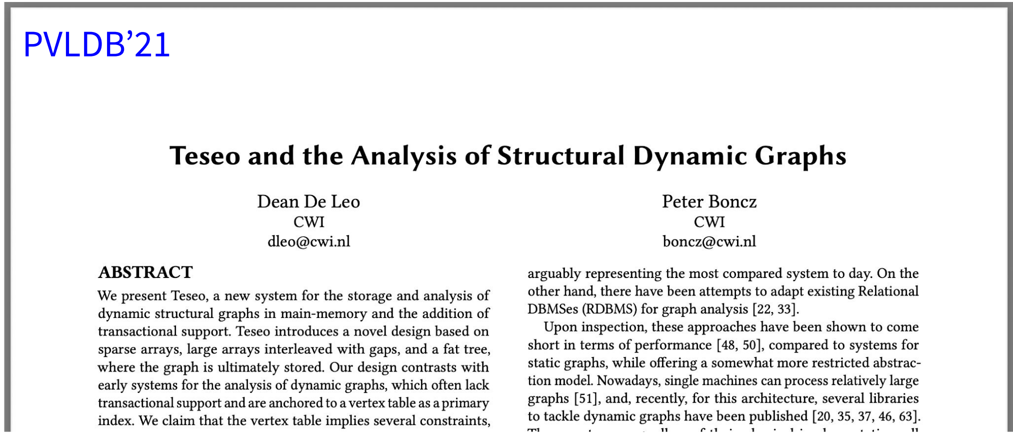
0 (subject, property, object) columns[1]. SQL systems tend to be more efficient than triple stores, because the latter need query plans with many self-joins – one per SPARQL

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) columna data structure, see Teseo



PVLDB'21

Teseo and the Analysis of Structural Dynamic Graphs

Dean De Leo
CWI
dleo@cwi.nl

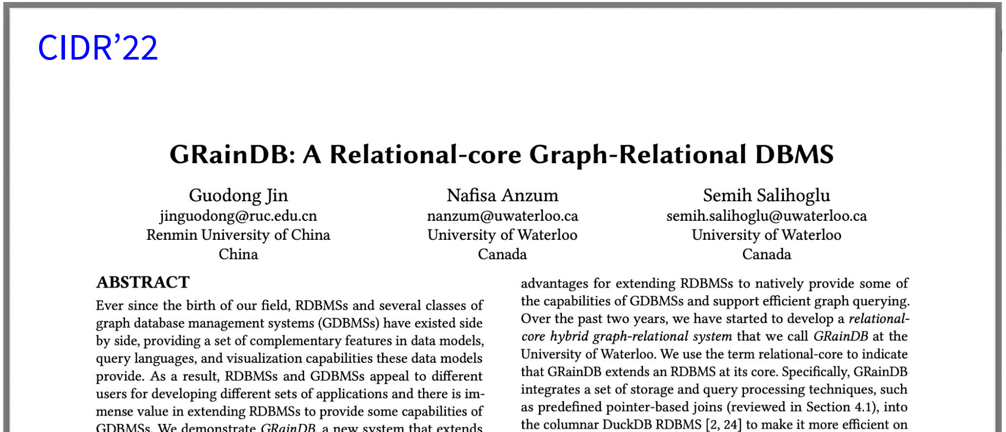
Peter Boncz
CWI
boncz@cwi.nl

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 (RDBMS) 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].



CIDR'22

GRainDB: A Relational-core Graph-Relational DBMS

Guodong Jin
jinguodong@ruc.edu.cn
Renmin University of China
China

Nafisa Anzum
nanzum@uwaterloo.ca
University of Waterloo
Canada

Semih Salihoglu
semih.salihoglu@uwaterloo.ca
University of Waterloo
Canada

ABSTRACT

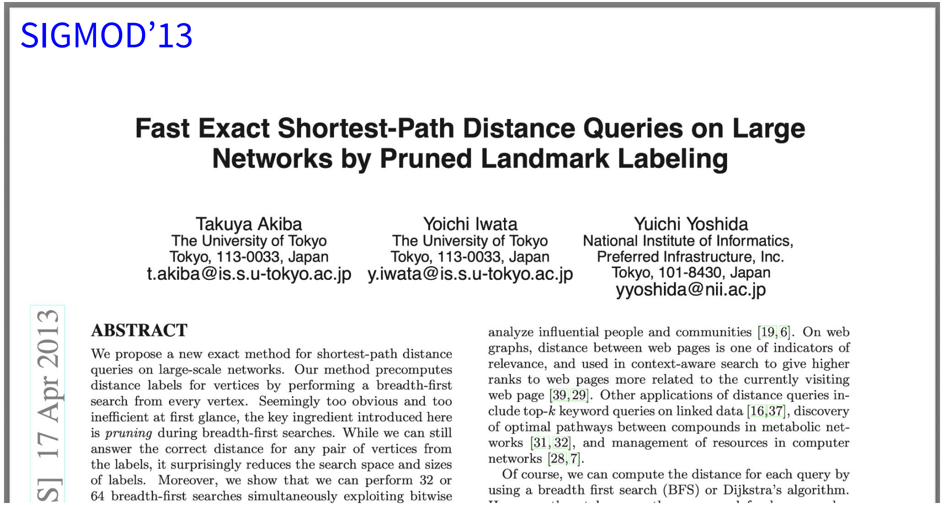
Ever since the birth of our field, RDBMSs and several classes of graph database management systems (GDBMSs) have existed side by side, providing a set of complementary features in data models, query languages, and visualization capabilities these data models provide. As a result, RDBMSs and GDBMSs appeal to different users for developing different sets of applications and there is immense value in extending RDBMSs to provide some capabilities of GDBMSs. We demonstrate *GRainDB*, a new system that extends

advantages for extending RDBMSs 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 DuckDB 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
Tokyo, 113-0033, Japan
t.akiba@is.s.u-tokyo.ac.jp

Yoichi Iwata
The University of Tokyo
Tokyo, 113-0033, Japan
y.iwata@is.s.u-tokyo.ac.jp

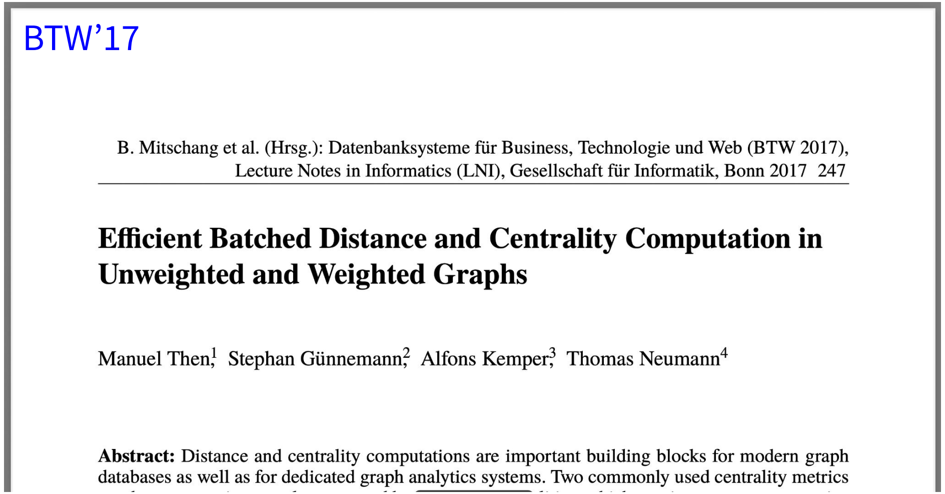
Yuichi Yoshida
National Institute of Informatics,
Preferred Infrastructure, Inc.
Tokyo, 101-8430, Japan
yyoshida@nii.ac.jp

ABSTRACT

We propose a new exact method for shortest-path distance queries on large-scale networks. Our method precomputes distance labels for vertices by performing a breadth-first search from every vertex. Seemingly too obvious and too inefficient at first glance, the key ingredient introduced here is *pruning* during breadth-first searches. While we can still answer the correct distance for any pair of vertices from the labels, it surprisingly reduces the search space and sizes of labels. Moreover, we show that we can perform 32 or 64 breadth-first searches simultaneously exploiting bitwise

analyze influential people and communities [19,6]. On web graphs, distance between web pages is one of indicators of relevance, and used in context-aware search to give higher ranks to web pages more related to the currently visiting web page [39,29]. Other applications of distance queries include top-*k* keyword queries on linked data [16,37], discovery of optimal pathways between compounds in metabolic networks [31,32], and management of resources in computer networks [28,7].

Of course, we can compute the distance for each query by using a breadth first search (BFS) or Dijkstra's algorithm.



BTW'17

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 247

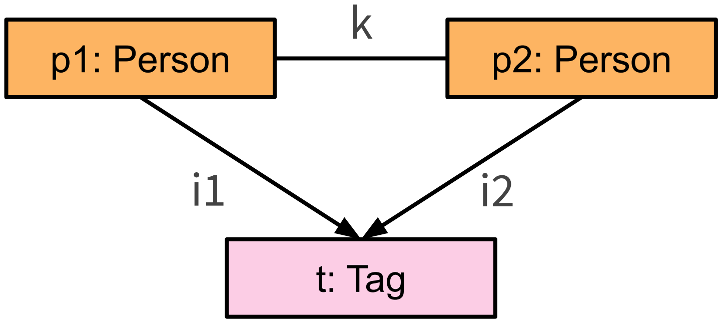
Efficient Batched Distance and Centrality Computation in Unweighted and Weighted Graphs

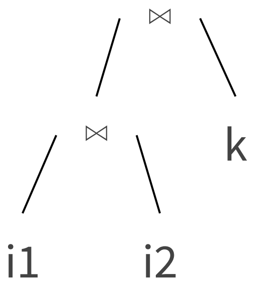
Manuel Then,[1] Stephan Günnemann,[2] Alfons Kemper,[3] Thomas Neumann[4]

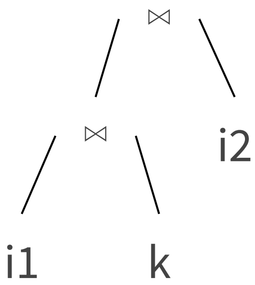
Abstract: Distance and centrality computations are important building blocks for modern graph databases as well as for dedicated graph analytics systems. Two commonly used centrality metrics

Complexity of subgraph matching

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

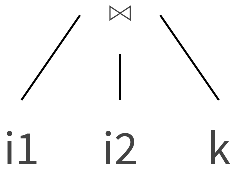






Triggered by many-to-many edges and skewed distributions.

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



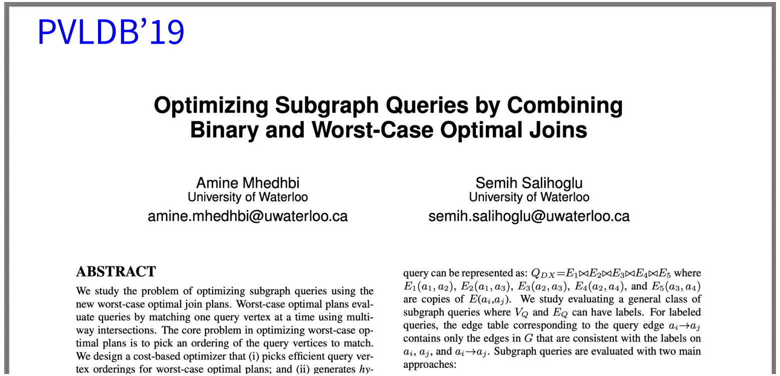
Research on Worst-Case Optimal Joins (WCOJ)

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)



PVLDB’19

Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins

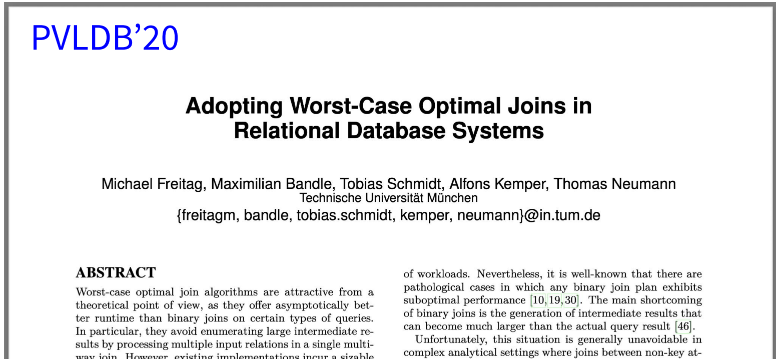
Amine Mhedhbi
University of Waterloo
amine.mhedhbi@uwaterloo.ca

Semih Salihoglu
University of Waterloo
semih.salihoglu@uwaterloo.ca

ABSTRACT

We study the problem of optimizing subgraph queries using the new worst-case optimal join plans. Worst-case optimal plans evaluate queries by matching one query vertex at a time using multiway intersections. The core problem in optimizing worst-case optimal plans is to pick an ordering of the query vertices to match. We design a cost-based optimizer that (i) picks efficient query vertex orderings for worst-case optimal plans; and (ii) generates *hy-*

query can be represented as: $Q_{DX}{=}E_1{\bowtie}E_2{\bowtie}E_3{\bowtie}E_4{\bowtie}E_5$ where $E_1(a_1,a_2)$, $E_2(a_1,a_3)$, $E_3(a_2,a_3)$, $E_4(a_2,a_4)$, and $E_5(a_3,a_4)$ are copies of $E(a_i,a_j)$. We study evaluating a general class of subgraph queries where V_Q and E_Q can have labels. For labeled queries, the edge table corresponding to the query edge $a_i{\rightarrow}a_j$ contains only the edges in G that are consistent with the labels on a_i, a_j, and $a_i{\rightarrow}a_j$. Subgraph queries are evaluated with two main approaches:



PVLDB’20

Adopting Worst-Case Optimal Joins in Relational Database Systems

Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, Thomas Neumann
Technische Universität München
{freitagm, bandle, tobias.schmidt, kemper, neumann}@in.tum.de

ABSTRACT

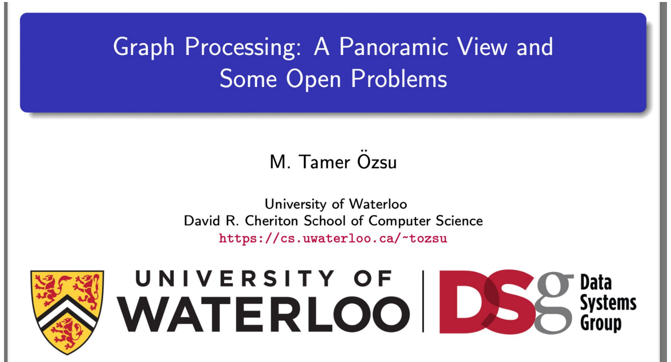
Worst-case optimal join algorithms are attractive from a theoretical point of view, as they offer asymptotically better runtime than binary joins on certain types of queries. In particular, they avoid enumerating large intermediate results by processing multiple input relations in a single multiway join. However, existing implementations incur a sizable

of workloads. Nevertheless, it is well-known that there are pathological cases in which any binary join plan exhibits suboptimal performance [10,19,30]. The main shortcoming of binary joins is the generation of intermediate results that can become much larger than the actual query result [46].

Unfortunately, this situation is generally unavoidable in complex analytical settings where joins between non-key at-

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
- …

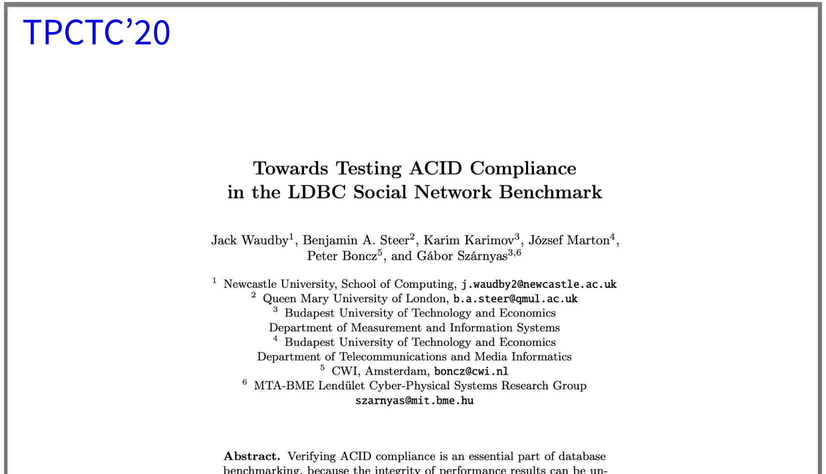


Graph Processing: A Panoramic View and Some Open Problems

M. Tamer Özsu

University of Waterloo
David R. Cheriton School of Computer Science
https://cs.uwaterloo.ca/~tozsu

UNIVERSITY OF WATERLOO | DSg Data Systems Group



TPCTC'20

Towards Testing ACID Compliance
in the LDBC Social Network Benchmark

Jack Waudby[1], Benjamin A. Steer[2], Karim Karimov[3], József Marton[4],
Peter Boncz[5], and Gábor Szárnyas[3,6]

[1] Newcastle University, School of Computing, j.waudby2@newcastle.ac.uk
[2] Queen Mary University of London, b.a.steer@qmul.ac.uk
[3] Budapest University of Technology and Economics
Department of Measurement and Information Systems
[4] Budapest University of Technology and Economics
Department of Telecommunications and Media Informatics
[5] CWI, Amsterdam, boncz@cwi.nl
[6] MTA-BME Lendület Cyber-Physical Systems Research Group
szarnyas@mit.bme.hu

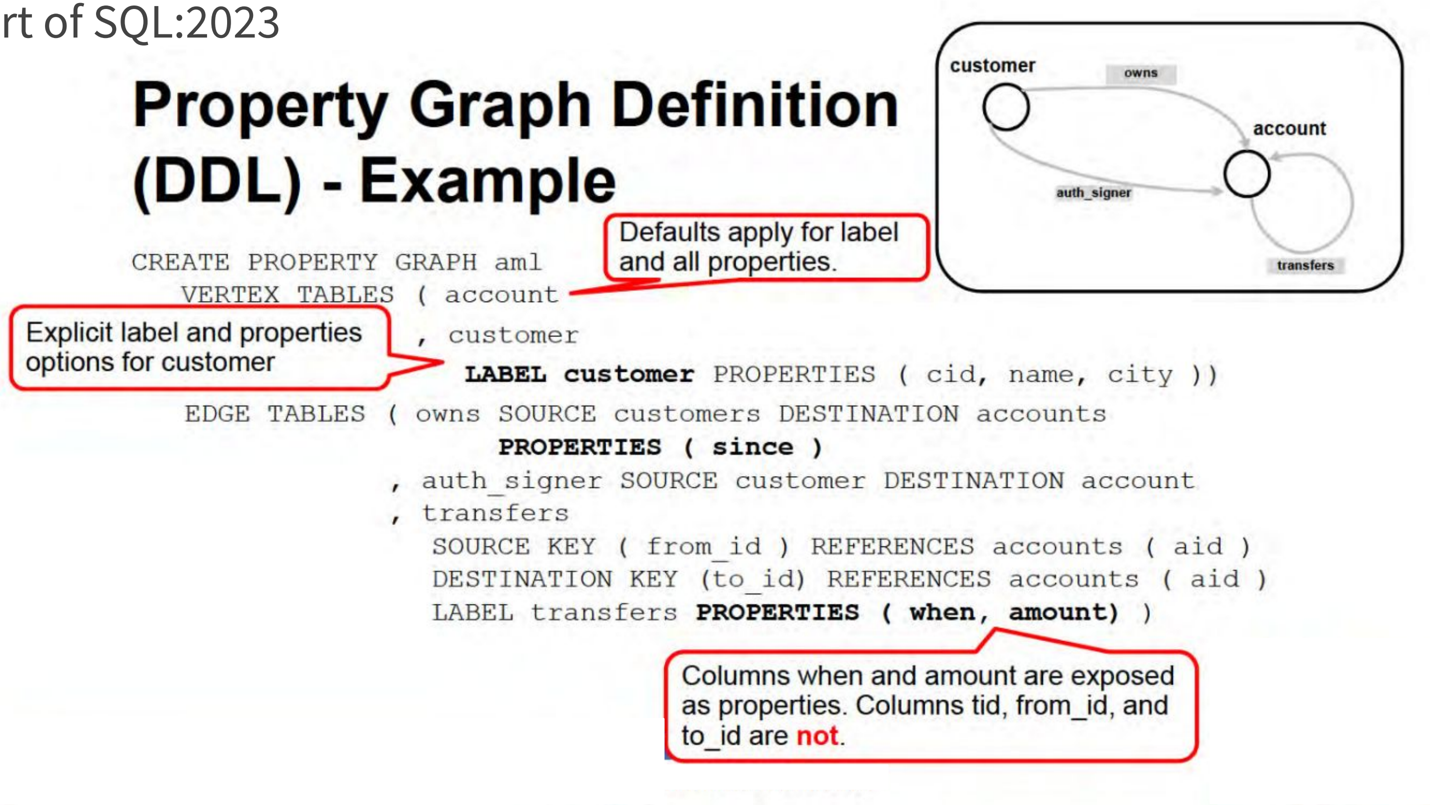
Abstract. Verifying ACID compliance is an essential part of database benchmarking, because the integrity of performance results can be un-

SQL:2023 aka SQL/PGQ

SQL/PGQ: CREATE PROPERTY GRAPH

Major part of SQL:2023

(slides)



Property Graph Definition (DDL) - Example

```
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) )
```

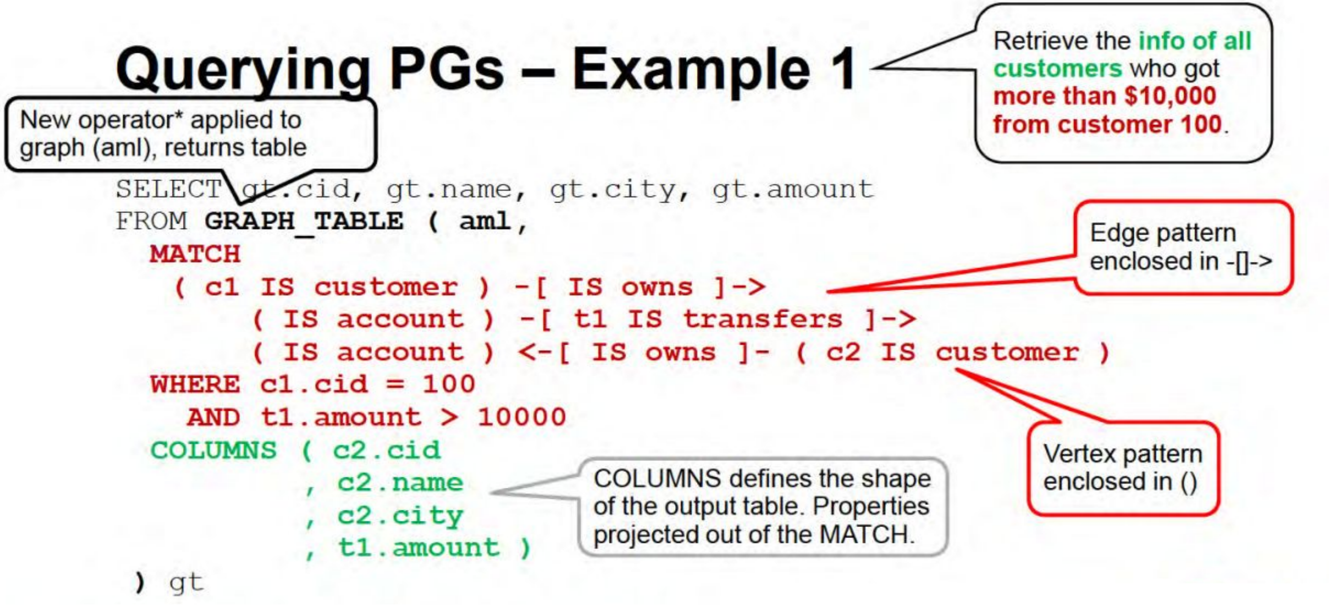
Defaults apply for label and all properties.

Explicit label and properties options for customer

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

Retrieve the info of all customers who got more than $10,000 from customer 100.

New operator* applied to graph (aml), returns table

```
SELECT gt.cid, gt.name, gt.city, gt.amount
FROM GRAPH_TABLE ( aml,
  MATCH
   ( c1 IS customer ) -[ IS owns ]->
        ( IS account ) -[ t1 IS transfers ]->
        ( IS account ) <-[ IS owns ]- ( c2 IS customer )
  WHERE c1.cid = 100
    AND t1.amount > 10000
  COLUMNS ( c2.cid
          , c2.name
          , c2.city
          , t1.amount )
 ) gt
```

Edge pattern enclosed in -[]->

Vertex pattern enclosed in ()

COLUMNS defines the shape of the output table. Properties projected out of the MATCH.

Access to ISO specs possible through liaison with LDBC. **Become an LDBC member!**

Graph Query Language (GQL)

New ISO standard with Cypher-like syntax:

```
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

LDBC
The graph & RDF benchmark reference

Gábor Szárnyas

Hannes Mühleisen
&
Mark Raasveldt