Graph Databases

Gábor Szárnyas
CWI, Amsterdam

5 November 2021

Main topics

What are graph databases?

How are they different from relational databases? (e.g. DuckDB, Postgres)

Why are they popular?

How could they work better?

What is a graph database (GDBMS)?

A graph database is a **specialized DBMS** that provides a **graph-oriented data model** and **an API optimized for graph operations**.

graph data model	graph API	graph visualization

neo4j	TigerGraph	Oracle Labs PGX	JanusGraph	Amazon Neptune
Cypher	GSQL	PGQL	Gremlin	Gremlin, SPARQL

Key GDBMS building blocks

⚫->⚫
property graph
data model

🌀
graph query
language

🔎
graph
visualization

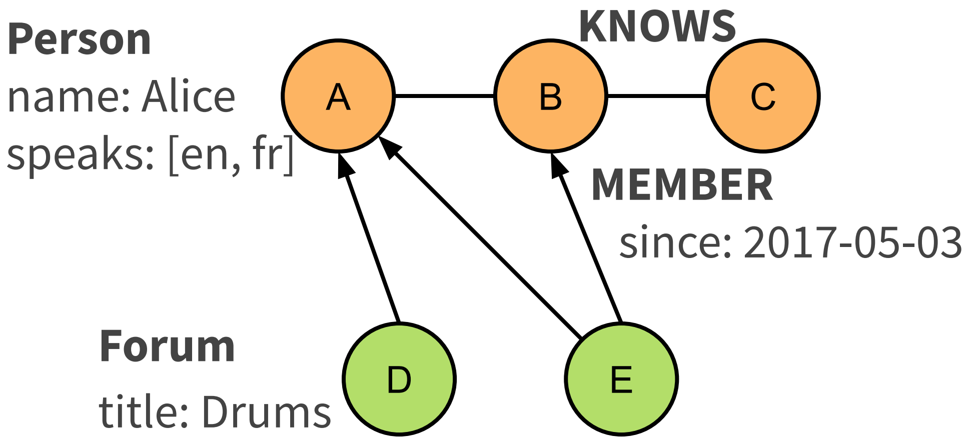
subgraph
matching

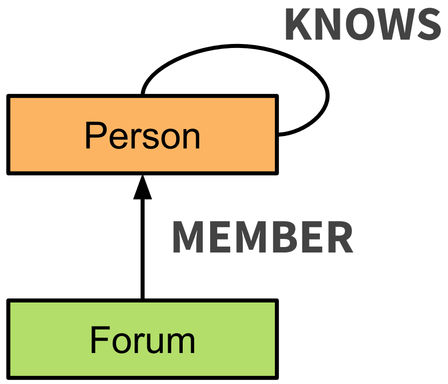
relational
queries

path
queries

stored
procedures

Data model: Property graph





The schema can be optional or mandatory. Lots of decisions and nuances -> [LDBC PGSWG]()

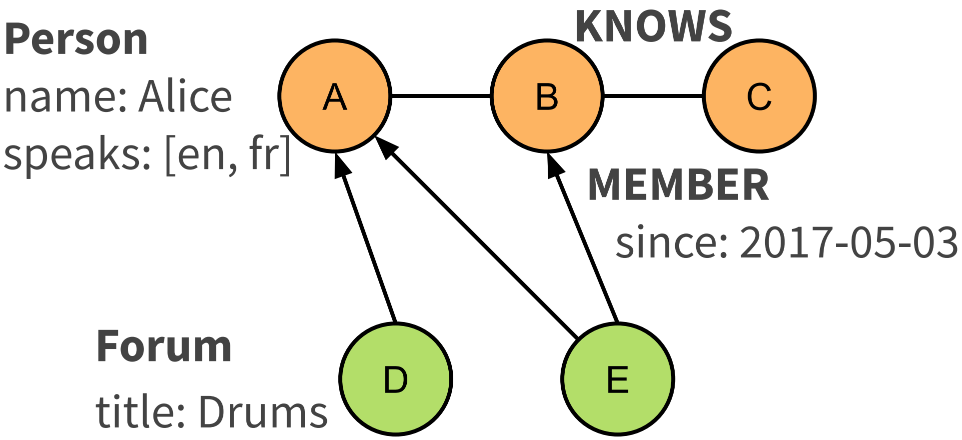
Query language: Cypher

```
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



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

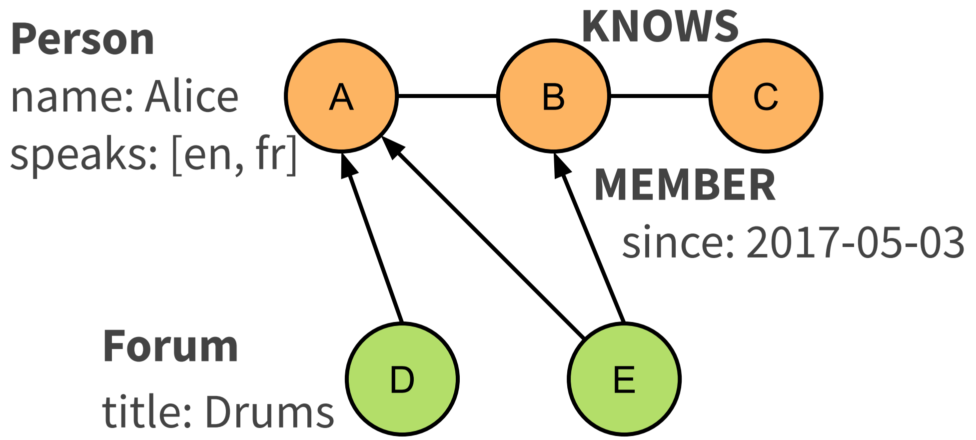
Query language: Cypher

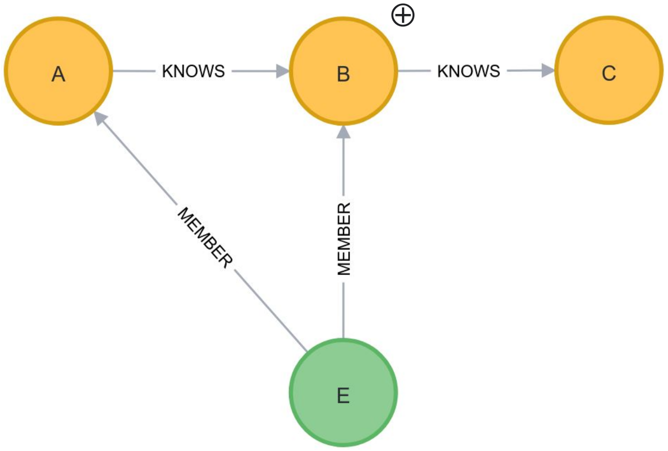
```
MATCH
  (p1:Person)-[:KNOWS]-(p2:Person),
  (p1)<-[:MEMBER]-(f:Forum)-[:MEMBER]->(p2),
  (p1)-[:KNOWS*]-(p3:Person)
WHERE NOT (f)-[:MEMBER]->(p3)
RETURN p1, f, count(p2), count(p3)
```

subgraph matching

path query (Kleene-star)

relational operators



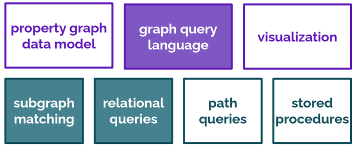


Graph Query Language (GQL)

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(*)
```

https://opencypher.org/articles/2019/09/12/SQL-and-now-GQL/
https://ldbcouncil.org/event/fourteenth-tuc-meeting/attachments/stefan-plantikow-gql.pdf



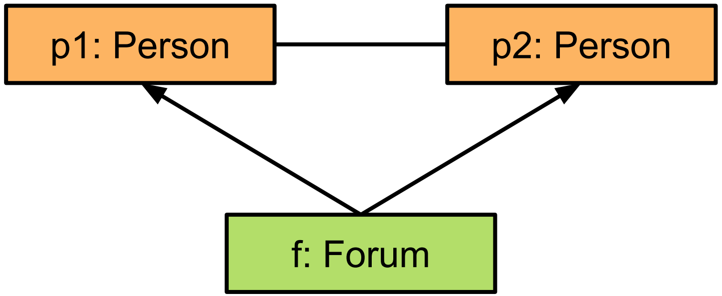
Pattern matching

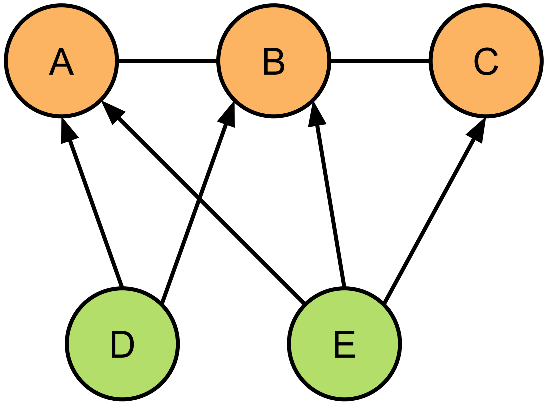
- **basic graph pattern**
- complex graph pattern
- recommendation query

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)

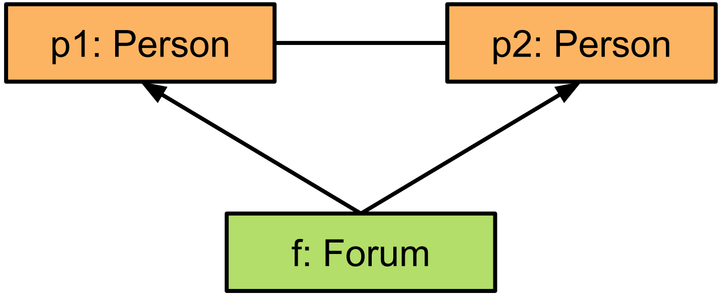
(A, B, E)

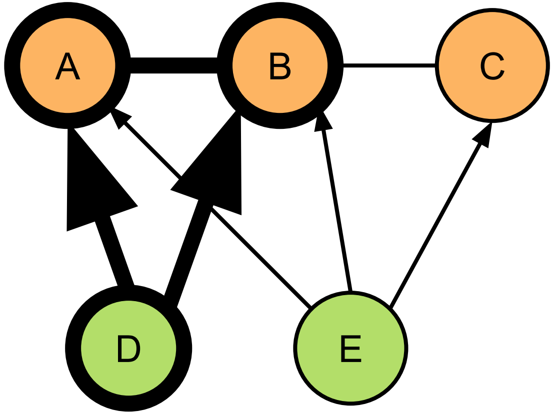
(B, C, E)

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)

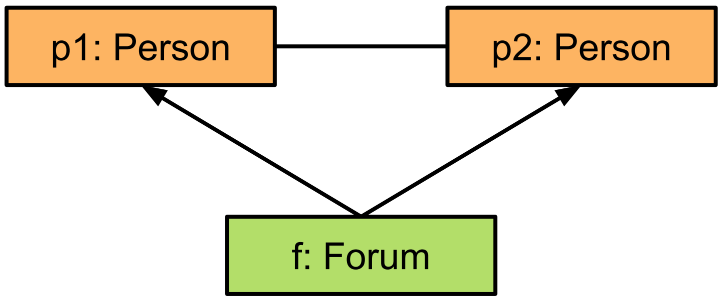
(A, B, E)

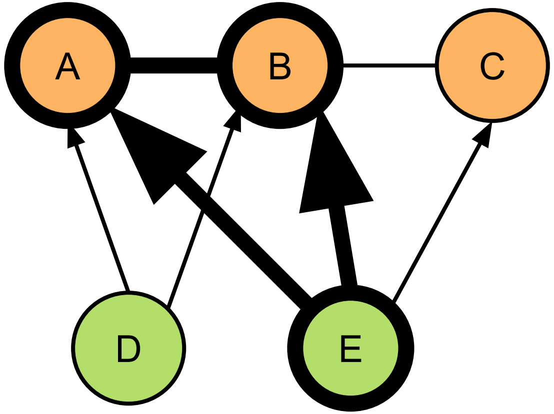
(B, C, E)

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)

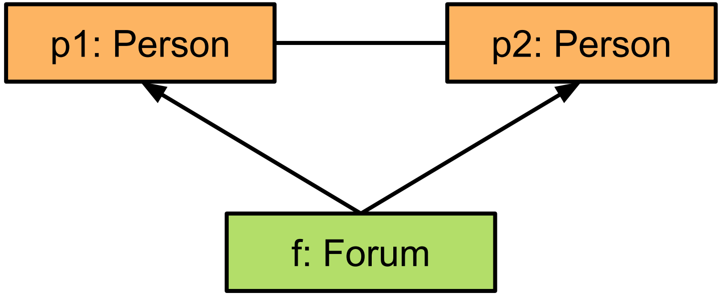
(A, B, E)

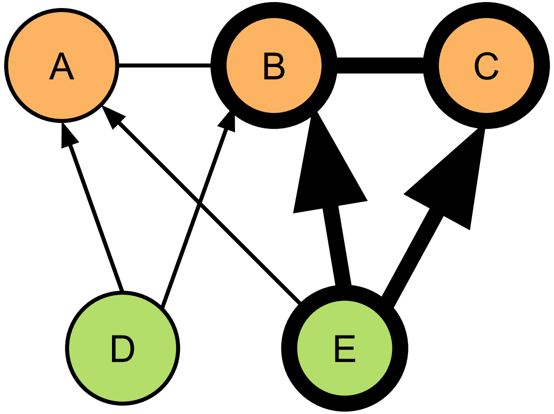
(B, C, E)

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)

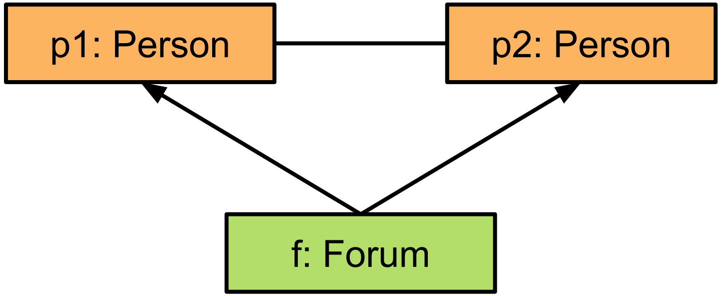
(A, B, E)

(B, C, E)

Subgraph matching (SQL)

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

Basic graph pattern: equijoins (SPJ)




```sql
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
```

Q: $m1 \bowtie m2 \bowtie knows$

property graph data model | graph query language | visualization

subgraph matching | relational queries | path queries | stored procedures

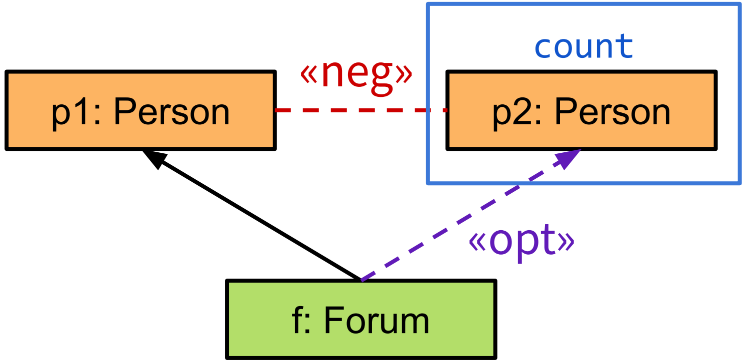
Pattern matching

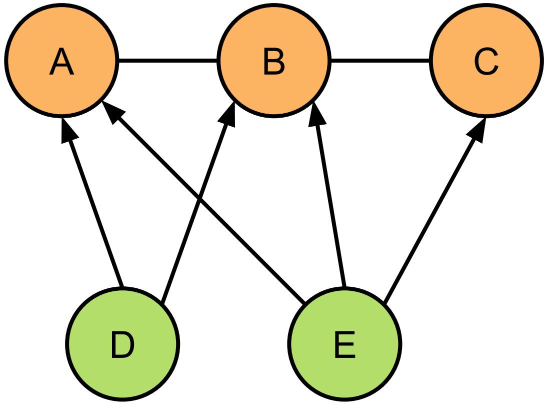
- basic graph pattern
- **complex graph pattern**
- recommendation query

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:

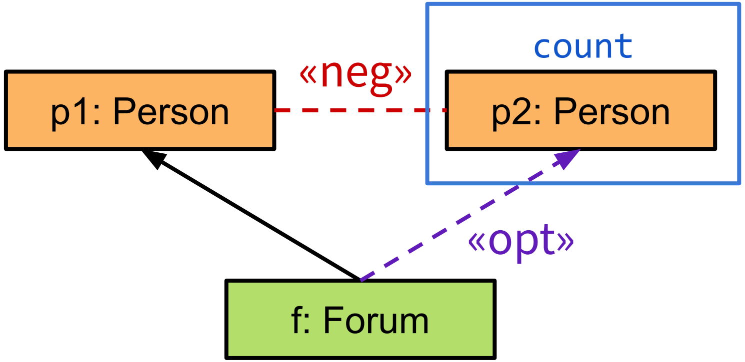
(D, 0)

(E, 1)

Subgraph matching (SQL)

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

Complex graph pattern: equijoins, outer joins, 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
```

γ (member ⟕ member ▷ knows)

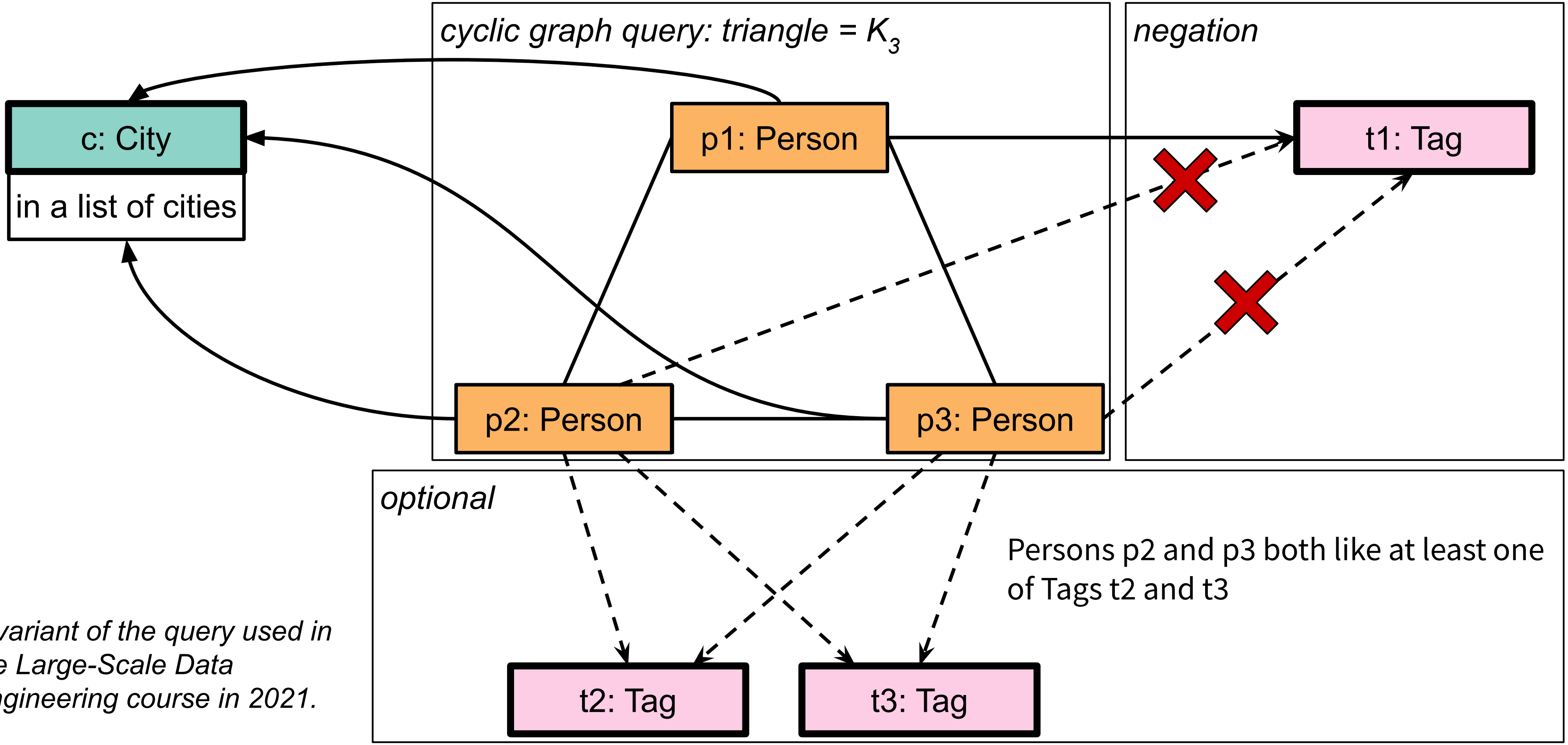
property graph data model | graph query language | visualization

subgraph matching | relational queries | path queries | stored procedures

Pattern matching

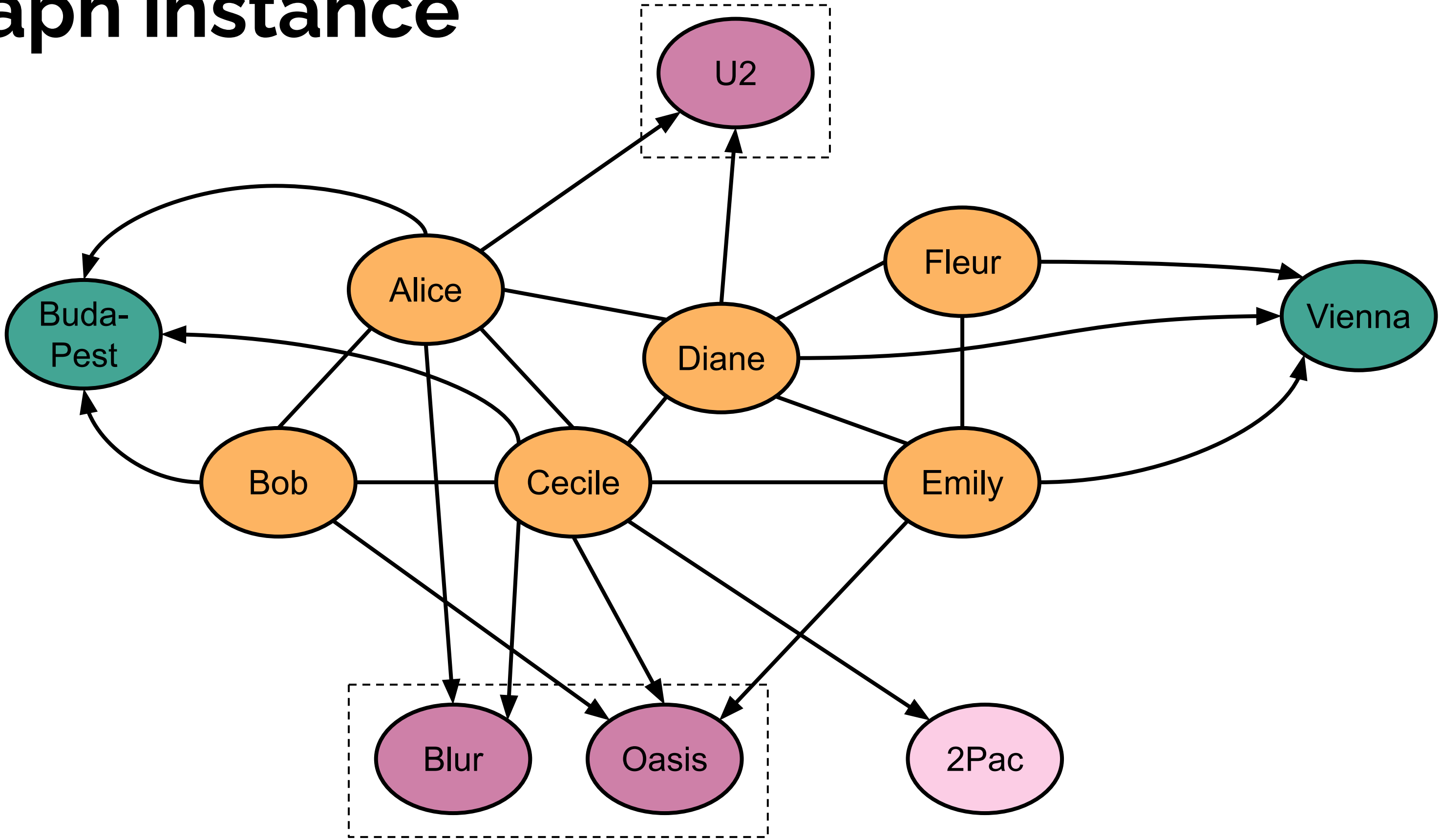
- basic graph pattern
- complex graph pattern
- **recommendation query**

Recommend concert tickets at a discount



A variant of the query used in the Large-Scale Data Engineering course in 2021.

Graph instance



Recommendation query in Cypher

```
MATCH (c:City),
 (p1:Person)-[:LIVES_IN]->(c),
 (p2:Person)-[:LIVES_IN]->(c),
 (p3:Person)-[:LIVES_IN]->(c),
 (p1)-[:KNOWS]-(p2)-[:KNOWS]-(p3)-[:KNOWS]-(p1),
 (p1)-[:LIKES]->(t1:Tag {name: 'U2'}),
 (t2:Tag {name: 'Blur'}),
 (t3:Tag {name: 'Oasis'})
WHERE c.name IN ['Budapest', 'Vienna']
 AND NOT (p2)-[:LIKES]->(t1)
 AND NOT (p3)-[:LIKES]->(t1)
 AND ((p2)-[:LIKES]->(t2) OR (p2)-[:LIKES]->(t3))
 AND ((p3)-[:LIKES]->(t2) OR (p3)-[:LIKES]->(t3))
 AND p1.id < p2.id
 AND p2.id < p3.id
RETURN p1.name, p2.name, p3.name
```

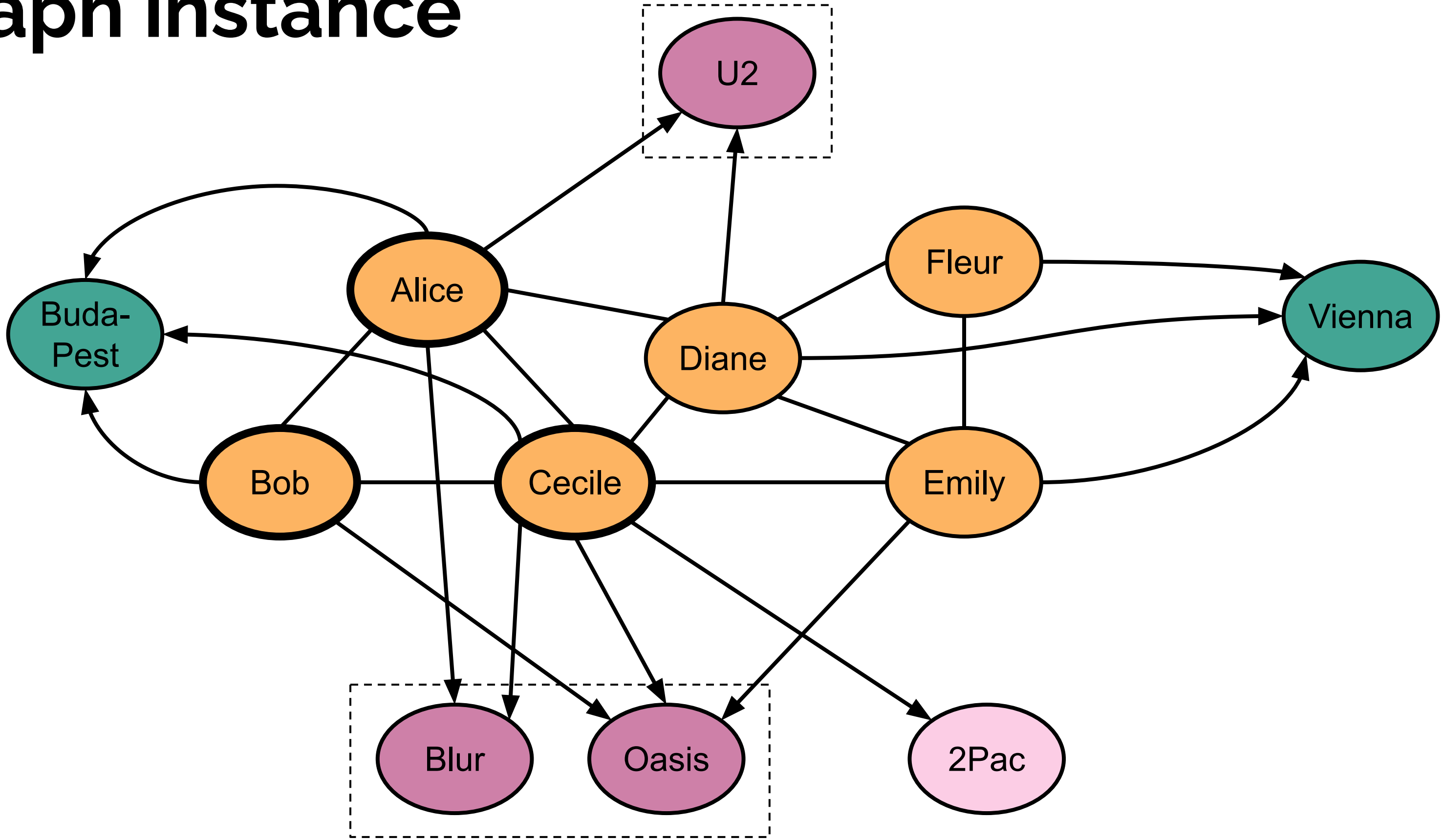
cyclic graph query: triangle = K_3

negation

optional

"p1.name"	"p2.name"	"p3.name"
"Alice"	"Bob"	"Cecile"

Graph instance



Recommendation query in SQL

```sql
SELECT DISTINCT p1.name, p2.name, p3.name
FROM City
JOIN Person p1  ON p1.livesin = City.id
JOIN Person p2  ON p2.livesin = City.id
JOIN Person p3  ON p3.livesin = City.id
JOIN knows k1   ON k1.person1id = p1.id
JOIN knows k2   ON k2.person1id = p2.id AND k2.person1id = k1.person2id
JOIN knows k3   ON k3.person1id = p3.id AND k3.person1id = k2.person2id AND k3.person2id = k1.person1id
JOIN Tag t1     ON t1.name = 'U2'
JOIN likes l1   ON l1.personid = p1.id AND l1.tagid = t1.id
JOIN Tag t2     ON t2.name = 'Blur'
JOIN Tag t3     ON t3.name = 'Oasis'
JOIN likes l2   ON l2.personid = p2.id AND (l2.tagid = t2.id OR l2.tagid = t3.id)
JOIN likes l3   ON l3.personid = p3.id AND (l2.tagid = t2.id OR l2.tagid = t3.id)
WHERE City.name IN ('Budapest', 'Vienna')
 AND NOT EXISTS (SELECT true FROM likes WHERE likes.personid = p2.id AND likes.tagid = t1.id)
 AND NOT EXISTS (SELECT true FROM likes WHERE likes.personid = p3.id AND likes.tagid = t1.id)
 AND p1.id < p2.id
 AND p2.id < p3.id
```

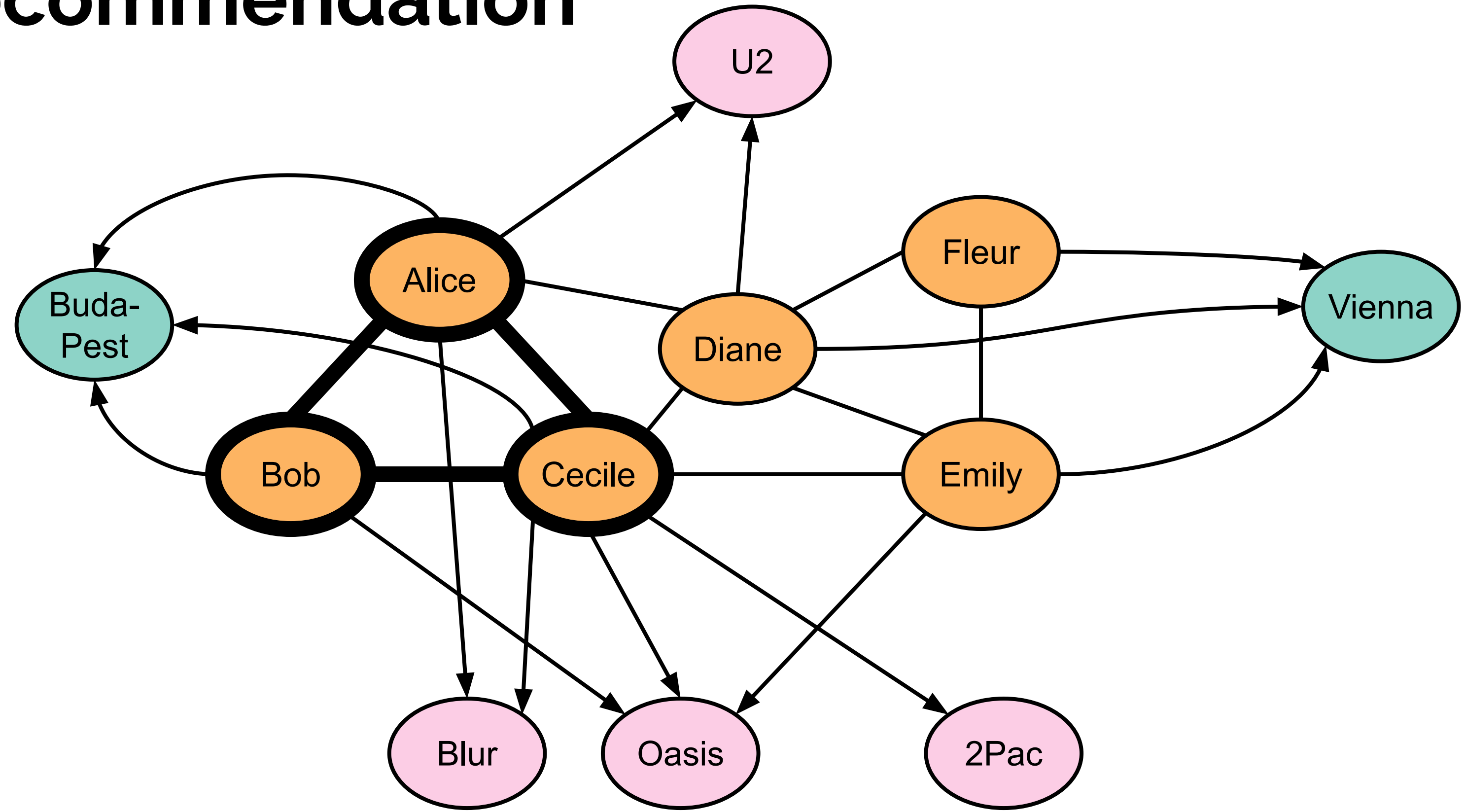
cyclic graph query: triangle = K_3

optional

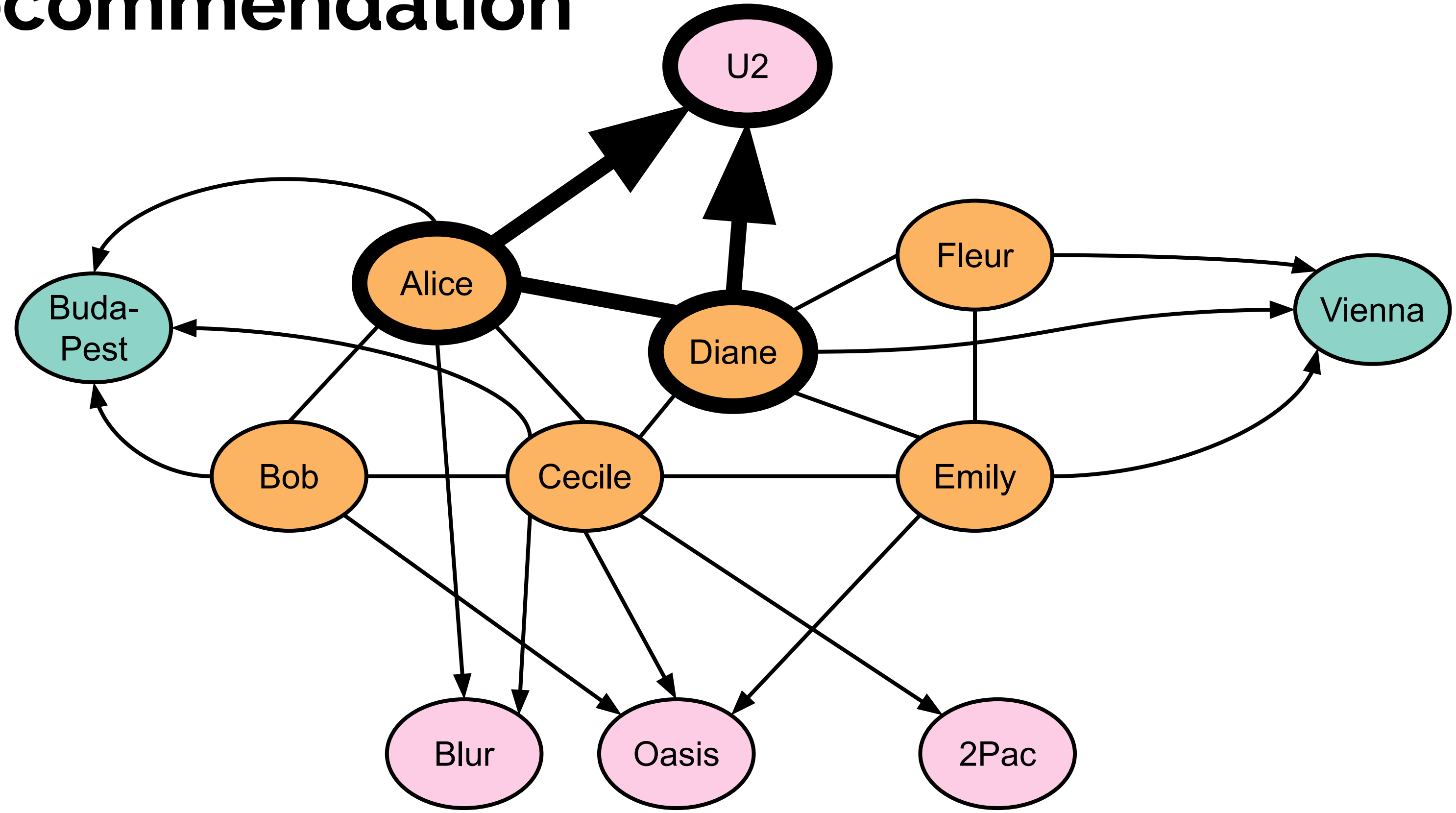
negation

Cypher query is approx. ½ of the SQL query

Recommendation

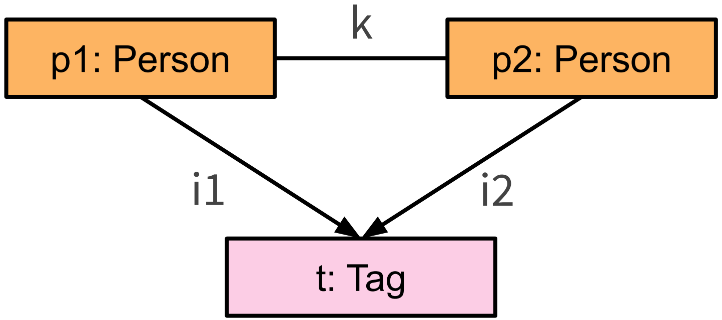


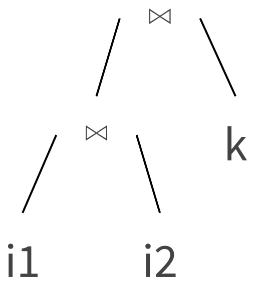
Recommendation

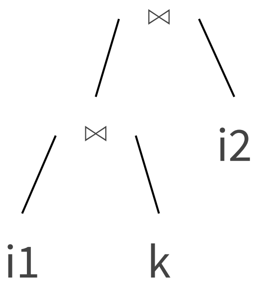


Complexity of subgraph matching

Subgraph isomorphism is NP-hard but *matching a given subgraph* is **polynomial**. Still, the complexity of evaluating **a triangle query with binary joins is provably suboptimal, O(|E|^2)**

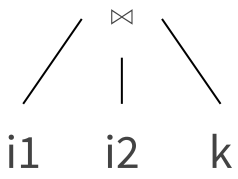






Triggered by many-to-many edges and skewed distributions.

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



Research on multiway joins

Subject to research in the last ~15 years:

- **FOCS’08** bounds on complexity
- **PODS’12** Generic-Join algorithm
- **SIGMOD’16** GraphflowDB demo
- **VLDB’19** query optimization in GraphflowDB
- **VLDB’20** integration into Umbra

Industry implementations: RelationalAI, XTDB

Recommendation is a main use case for GDBMSs

UWaterloo Survey (2017, 2020):

The second most popular application was the use of a graph-based application data to personalize user interactions and provide better recommendations for the customers of a business.

Reason for popularity: ? (maybe visualization?)

property graph data model | graph query language | visualization

subgraph matching | relational queries | path queries | stored procedures

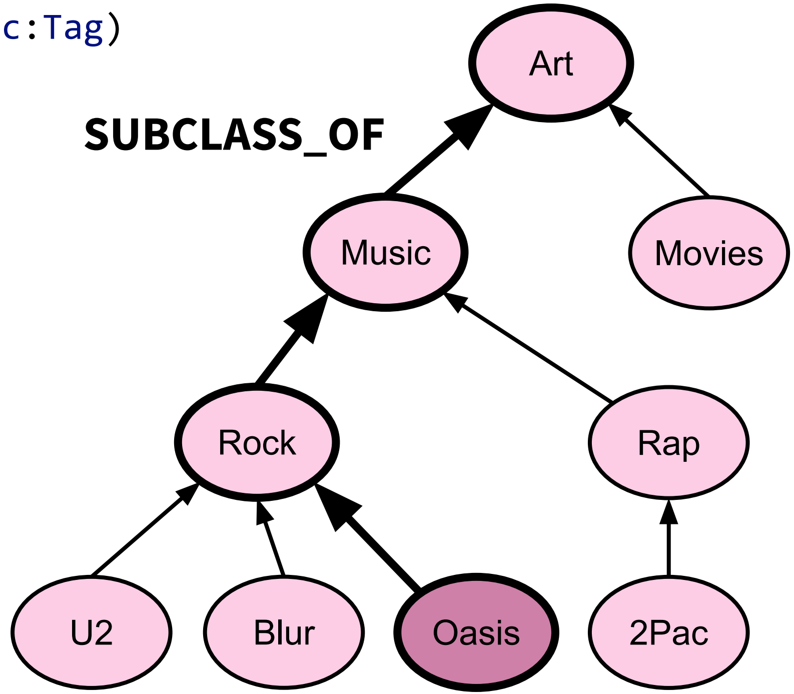
Path queries

- **tree query**
- unweighted path query
- weighted shortest path query
- connected components

Tree query: Cypher

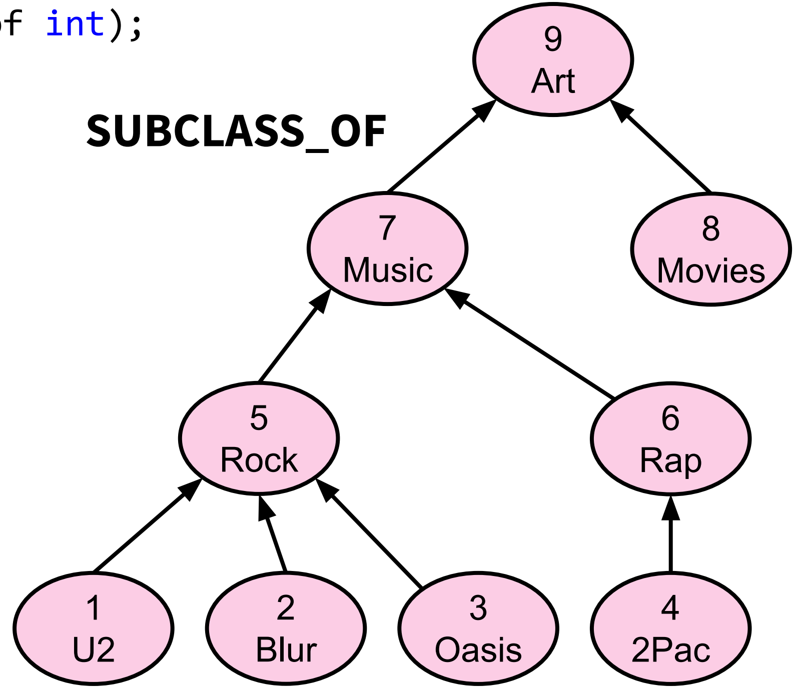
```
MATCH (:Tag {name: 'Oasis'})-[:SUBCLASS_OF*]->(sc:Tag)
RETURN sc.name AS superclass
```

```
superclass
----------
Rock
Music
Art
```



Tree query: SQL data

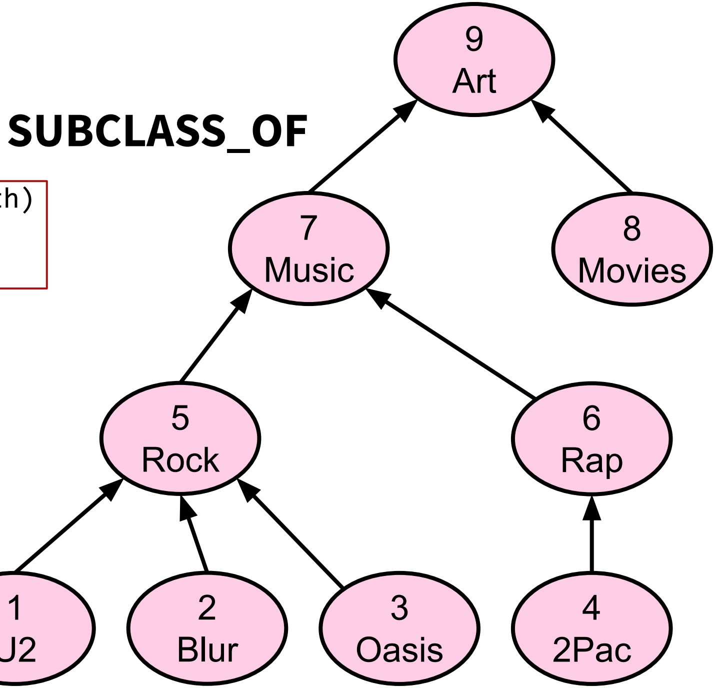
```sql
CREATE TABLE tag(id int, name varchar, subclassof int);

INSERT INTO tag VALUES
 (1, 'U2',      5),
 (2, 'Blur',    5),
 (3, 'Oasis',   5),
 (4, '2Pac',    6),
 (5, 'Rock',    7),
 (6, 'Rap',     7),
 (7, 'Music',   9),
 (8, 'Movies',  9),
 (9, 'Art',     NULL)
;
```



Tree query with string concat

```
WITH RECURSIVE tag_hierarchy(id, source, path) AS (
 SELECT id, name, name AS path
 FROM tag
 WHERE subclassof IS NULL
UNION ALL
 SELECT tag.id, tag.name, (tag.name || ' -> ' || tag_hierarchy.path)
 FROM tag, tag_hierarchy
 WHERE tag.subclassof = tag_hierarchy.id
)
SELECT source, path
FROM tag_hierarchy;
```

source	path
Art	Art
Music	Music -> Art
Movies	Movies -> Art
Rock	Rock -> Music -> Art
Rap	Rap -> Music -> Art
U2	U2 -> Rock -> Music -> Art
Blur	Blur -> Rock -> Music -> Art
Oasis	Oasis -> Rock -> Music -> Art
2Pac	2Pac -> Rap -> Music -> Art

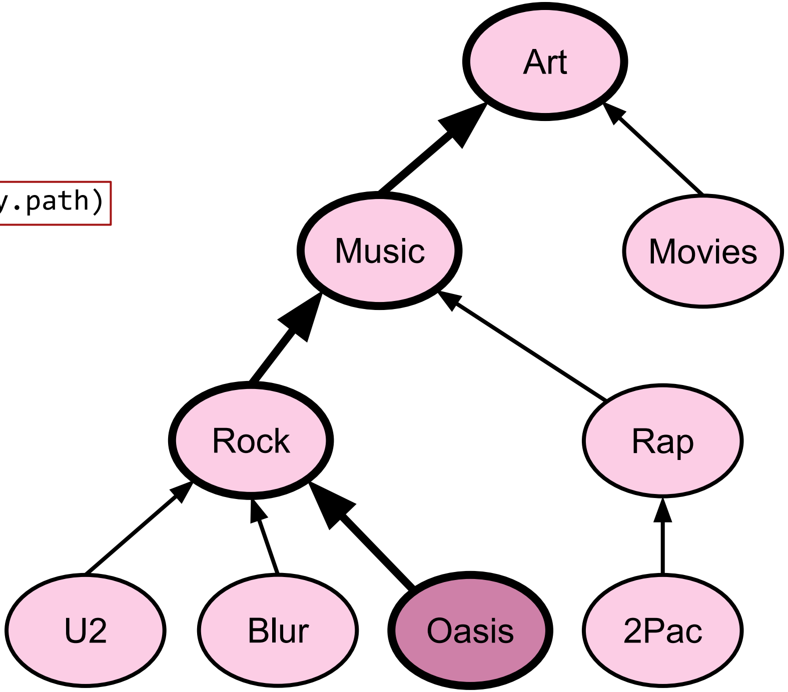


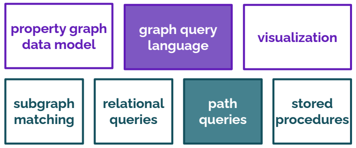
Tree query with list concat

DuckDB 0.3.1-dev
🏆 to Laurens

```sql
WITH RECURSIVE tag_hierarchy(id, source, path) AS (
 SELECT id, name, [name] AS path
 FROM tag
 WHERE subclassof IS NULL
UNION ALL
 SELECT tag.id, tag.name, list_prepend(tag.name, tag_hierarchy.path)
 FROM tag, tag_hierarchy
 WHERE tag.subclassof = tag_hierarchy.id
)
SELECT source, unnest(path) AS superclass
FROM tag_hierarchy
WHERE source = 'Oasis';
```

source	superclass
Oasis	Oasis
Oasis	Rock
Oasis	Music
Oasis	Art



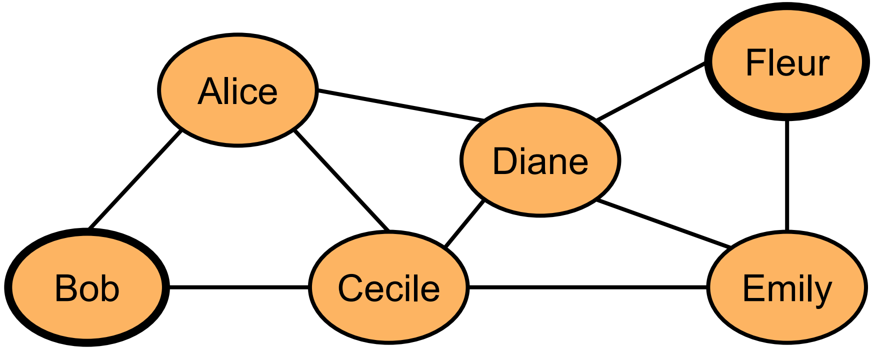


Path queries

- tree query
- **unweighted path query**
- weighted shortest path query
- connected components

Unweighted shortest path in Cypher

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

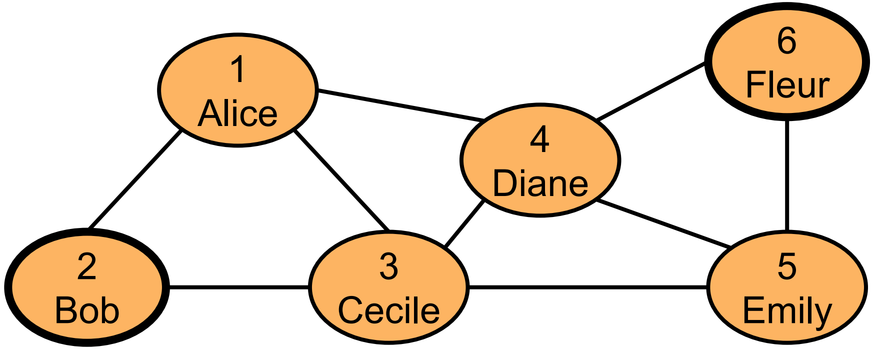


Result:

```
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

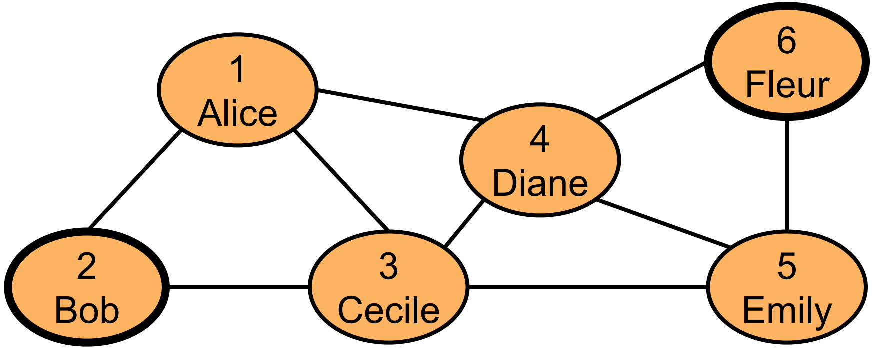
id [PK]	name
1	Alice
2	Bob
3	Cecile
4	Diane
5	Emily
6	Fleur

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

```
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 () 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 () 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 () 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 () 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;
```

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 () 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 () 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 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 () 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 () 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

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 () 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 () 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';
```

cycle detection

Unweighted shortest path query in Postgres 14

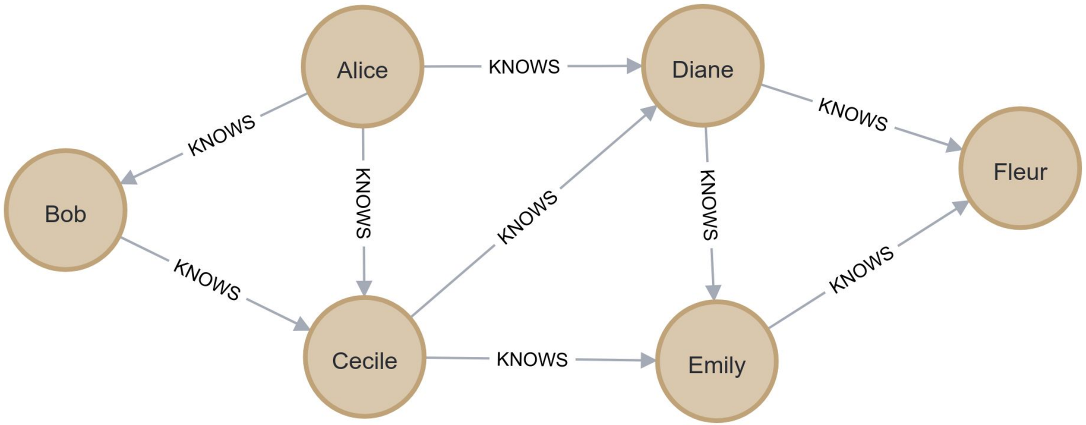
```sql
WITH RECURSIVE paths(startPerson, endPerson, path, level, endPersonReached) AS (
  SELECT person1id AS startPerson, person2id AS endPerson,
         ARRAY[person1id, person2id]::bigint[] AS path, 1 AS level,
         max(CASE WHEN p2.name = 'Fleur' THEN 1 ELSE 0 END) OVER () AS endPersonReached
    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 1 ELSE 0 END) OVER () AS endPersonReached
    FROM paths
    JOIN knows     ON paths.endPerson = knows.person1id
    JOIN Person p2 ON p2.id = knows.person2id
   WHERE paths.endPersonReached = 0
) CYCLE endPerson SET is_cycle TO true DEFAULT false USING cycle_path
SELECT path, level, endPersonReached AS epr, is_cycle
FROM paths
JOIN Person ON Person.id = paths.endPerson
WHERE Person.name = 'Fleur'
 AND is_cycle = false;
```

cycle detection

```
   path     | level | epr | is_cycle
------------+-------+-----+----------
 {2,3,5,6}  |     3 |   1 | f
 {2,3,4,6}  |     3 |   1 | f
 {2,1,4,6}  |     3 |   1 | f
(3 rows)
```

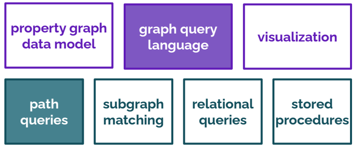
Visualize all unweighted shortest paths

```
MATCH p=allShortestPaths(
  (p1:Person {name: 'Bob'})-[:KNOWS*]-(p2:Person {name: 'Fleur'}))
RETURN p
```




```
[p IN nodes(p) | p.name] AS persons
```

"persons"
["Bob","Cecile","Emily","Fleur"]
["Bob","Cecile","Diane","Fleur"]
["Bob","Alice","Diane","Fleur"]



Path queries

- tree query
- unweighted path query
- **weighted shortest path query**
- connected components

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 algorithm

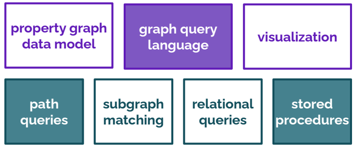
- Unweighted shortest path -> query
- Weighted shortest path -> algorithm

Cheapest path (proposal)

Weighted shortest path -> cheapest path.

```
MATCH p AS CHEAPEST (c1 IS customer)-[t IS TRANSFERS COST t.amount]->*(c2 IS customer)
WHERE c1.cid = :customer1Id
  AND c2.cid = :customer2Id
RETURN p
```

(Proposed to Cypher and GQL)



Path queries

- tree query
- unweighted path query
- weighted shortest path query
- **connected components**

Connected components

Can be formulated as a path query but will not be efficient

```
MATCH p=shortestPath((p1:Person)-[:KNOWS*]-(p2:Person))
WHERE p1 <> p2
RETURN p1.id AS node, count(p2.id)+1 AS componentSize
```

The compiler cannot figure out that this is a CC problem -> stored procedures are better

Connected components algorithm in SQL

[In-database connected component analysis](#) (ICDE 2020), implemented in Apache HAWQ

```
r.log_exec("setup", 0, """\
create table ccgraph as
    select v1, v2 from {0}
    union all
    select v2, v1 from {0}
    distributed by (v1);
""".format(dataset))

roundno = 0
stackA = []
stackB = []
while True:
    roundno += 1
    ccreps = "ccreps{}".format(roundno)
    r_A = 0
    while r_A == 0:
        r_A = random.randint(-2**63,2**63-1)
    r_B = random.randint(-2**63,2**63-1)
    stackA.append(r_A)
    stackB.append(r_B)

    r.log_exec("ccreps", roundno, """\
    create table {ccreps} as
        select v1 v,
            least(axplusb({A},v1,{B}),
                min(axplusb({A},v2,{B}))) rep
        from ccgraph
        group by v1
        distributed by (v);
    """.format(ccreps=ccreps, A=r_A, B=r_B))
```

ax+b UDF

```
    r.log_exec("ccgraph2", roundno, """\
    create table ccgraph2 as
        select r1.rep as v1, v2
        from ccgraph, {} as r1
        where ccgraph.v1 = r1.v
        distributed by (v2);
    """.format(ccreps))
    r.log_drop("ccgraph")

    graphsize = r.log_exec("ccgraph3", roundno, """\
    create table ccgraph3 as
        select distinct v1, r2.rep as v2
        from ccgraph2, {} as r2
        where ccgraph2.v2 = r2.v
            and v1 != r2.rep
        distributed by (v1);
    """.format(ccreps))
    r.log_drop("ccgraph2")
    r.execute("alter table ccgraph3 rename to ccgraph")

    if graphsize == 0:
        break
```

```
accA = 1
accB = 0

while True:
    roundno -= 1
    (accA, accB) = (r.axplusb(accA, stackA.pop(),0),
                    r.axplusb(accA, stackB.pop(), accB))
    if roundno == 0:
        break
    ccrepsr = "ccreps{}".format(roundno)
    ccrepsr1 = "ccreps{}".format(roundno+1)
    r.log_exec("result", roundno, """\
    create table tmp as
        select r1.v as v,
            coalesce(r2.rep,axplusb({A},r1.rep,{B})) as rep
        from {r1} as r1 left outer join
            {r2} as r2
            on (r1.rep=r2.v)
        distributed by (v);
    """.format(A=accA, B=accB, r1=ccrepsr, r2=ccrepsr1))
    r.log_drop(ccrepsr)
    r.log_drop(ccrepsr1)
    r.execute("alter table tmp rename to {}".format(ccrepsr))

r.execute("alter table ccreps1 rename to ccresult")
r.log_drop("ccgraph")
```

Fig. 8. Our implementation of Randomised Contraction in Python/SQL.

Index-free adjacency

“Relational databases can’t join efficiently”

It’s a widespread sentiment in the graph DB community that RDBMSs are inefficient.

A slide from the deck [“Dgraph Series A 2019”]
by Manish R. Jain

> **Pain**
>
> - Joins and traversals in SQL are slow.

A [tweet by Frank McSherry]
(influential OS/DB researcher)
about *what he heard* at an
industry graph conference.

> **Frank McSherry**
> @frankmcsherry
>
> Replying to @frankmcsherry
>
> Have now talked with many people who believe that graphDBs are new and neat because RDBMSs obvs use full table scans.

Defining a Graph Database

(Marko Rodriguez, AT&T's graph systems architect, 2010)

A graph database is any storage system that provides index-free adjacency.[23][24]

[23] There is no "official" definition of what makes a database a graph database. The one provided is my definition (respective of the influence of my collaborators in this area). However, hopefully the following argument will convince you that this is a necessary definition. Given that any database can model a graph, such a definition would not provide strict enough bounds to yield a formal concept (i.e. $\top$).

[24] There is adjacency between the elements of an index, but if the index is not the primary data structure of concern (to the developer), then there is indirect/implicit adjacency, not direct/explicit adjacency. A graph database exposes the graph as an explicit data structure (not an implicit data structure).

Neighbourhood lookup

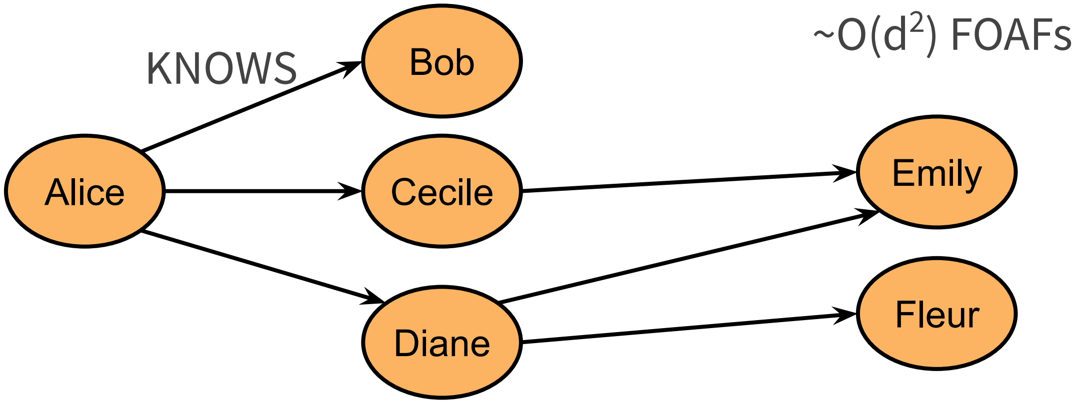
Efficient neighbourhood lookups (fetch all neighbours of node x) are important for

- traversals (multi-source BFS, bidirectional BFS)
- pattern matching (many WCOJ algorithms intersect neighbourhoods)

Pointers

Lookup is quick when traversing a “many-to-one” edge.

But the expensive traversals go on “many-to-many” edges:



Tradeoffs of pointers

Pointers lead to:

- memory fragmentation
- contention on allocation (synchronization between threads)
- large memory overhead for every node/edge (~2x)
- no compression (another ~4x)
- “tuple-at-a-time” execution with random access at every step

Better solution:

- buffer manager
- ordered tabular storage w/ compression
- vectorized execution (e.g. vectors of 64 tuples, cache misses happen at once)

Columnar store solution

- Everything from an RDBMS
- A graph data structure that supports **analytical queries** and **updates**

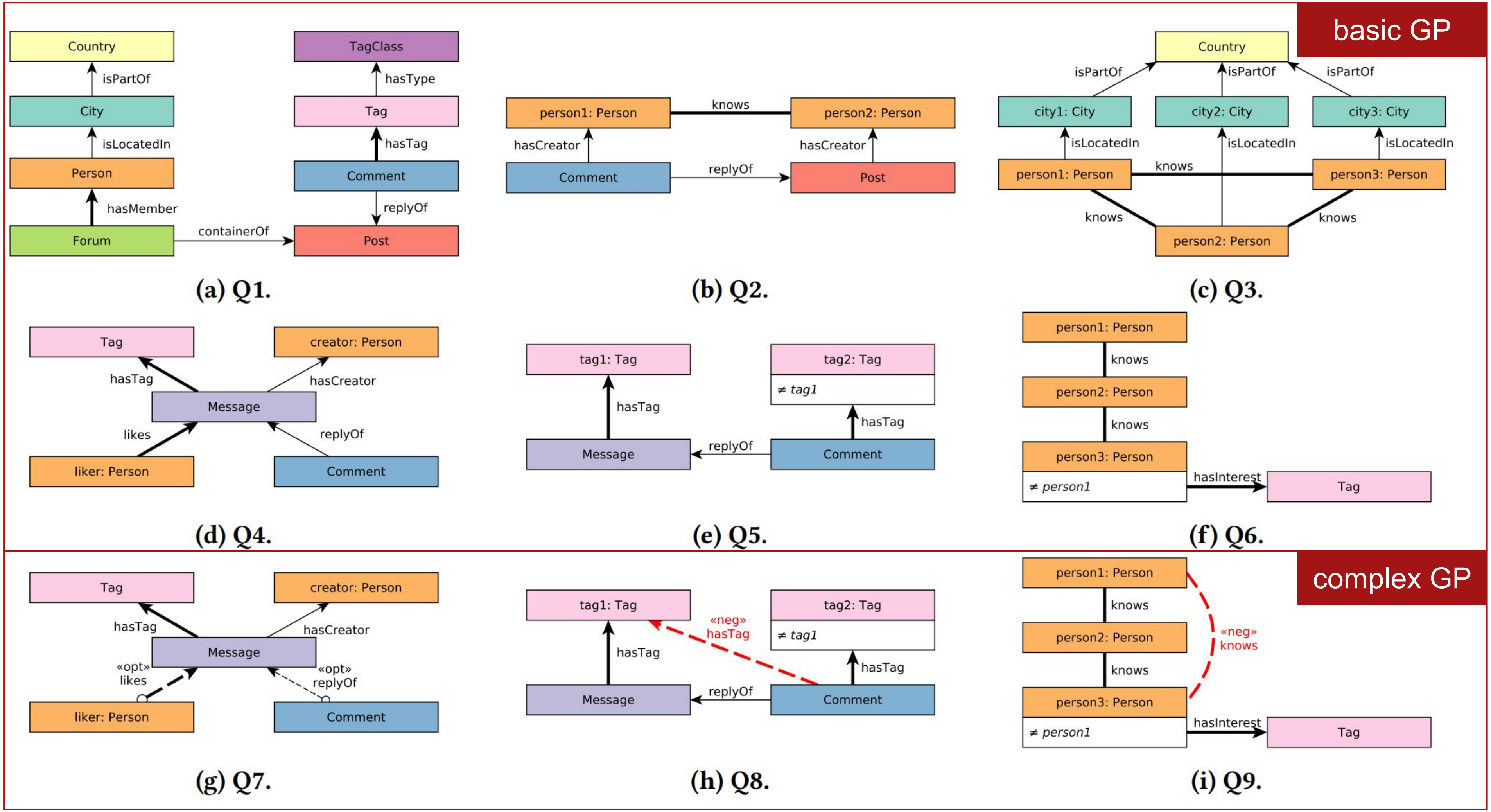
Analytical queries benefit from positional indexing (~pointer-based lookups)

- Invisible join, also supported by DuckDB (Diego Tomé)
- Teseo: packed memory arrays for an updatable CSR-like data structure (Dean de Leo)
- Sortledton: adjacency lists with fast intersection support (Per Fuchs)

Pattern matching performance

Published at GRADES-NDA 2021, Mhedhbi et al.

Designed to prove that Hannes is wrong



(a) Q1. (b) Q2. (c) Q3.

(d) Q4. (e) Q5. (f) Q6.

(g) Q7. (h) Q8. (i) Q9.

Relational database performance

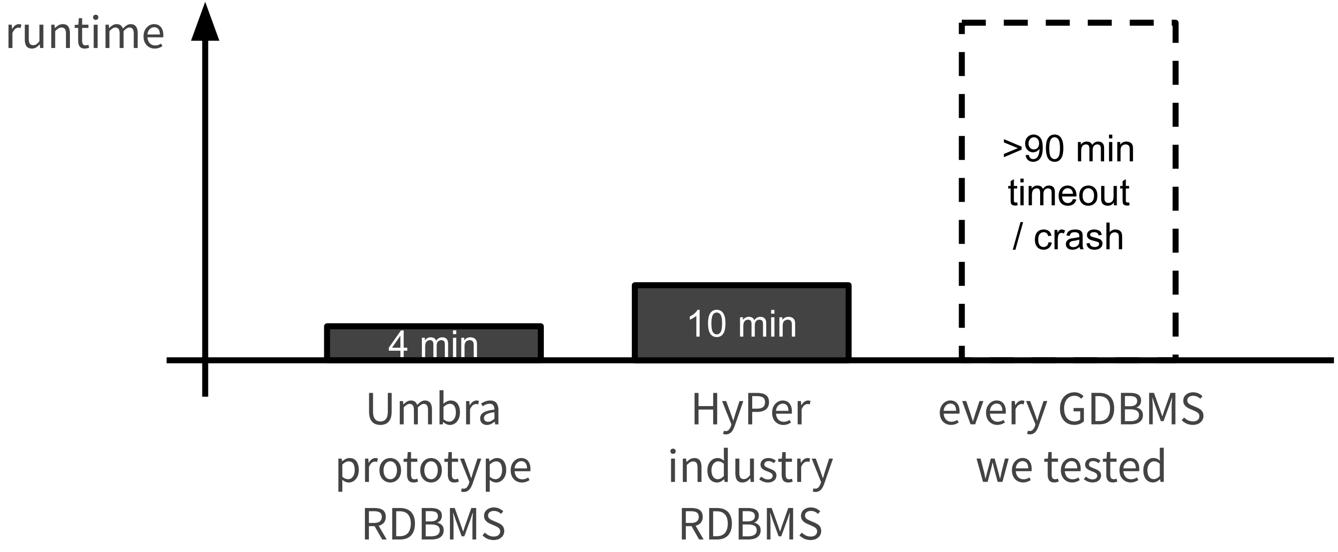
Best relational systems: columnar stores with compression

- HyPer (industry system, 2010-)
- Umbra: worst-case optimal join support (prototype, 2020-)

Benchmark environment: cloud VM, 370GB RAM, 48 vCPUs

GDBMS performance for subgraph queries

- Load the data: 100M vertices, 650M edges
- Run all 9 queries one-by-one
- Environment: cloud VM, 370GB RAM, 48 vCPU cores



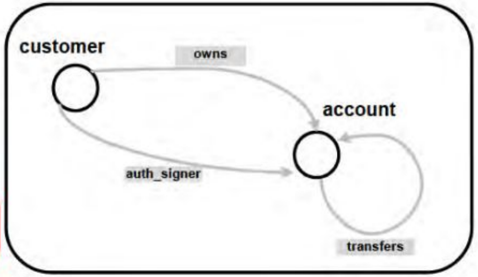
Building a GDBMS on an RDBMS

SQL/PGQ: CREATE PROPERTY GRAPH

To be released in 2022

(slides)

Property Graph Definition (DDL) - Example



Defaults apply for label and all properties.

Explicit label and properties options for customer

```
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) )
```

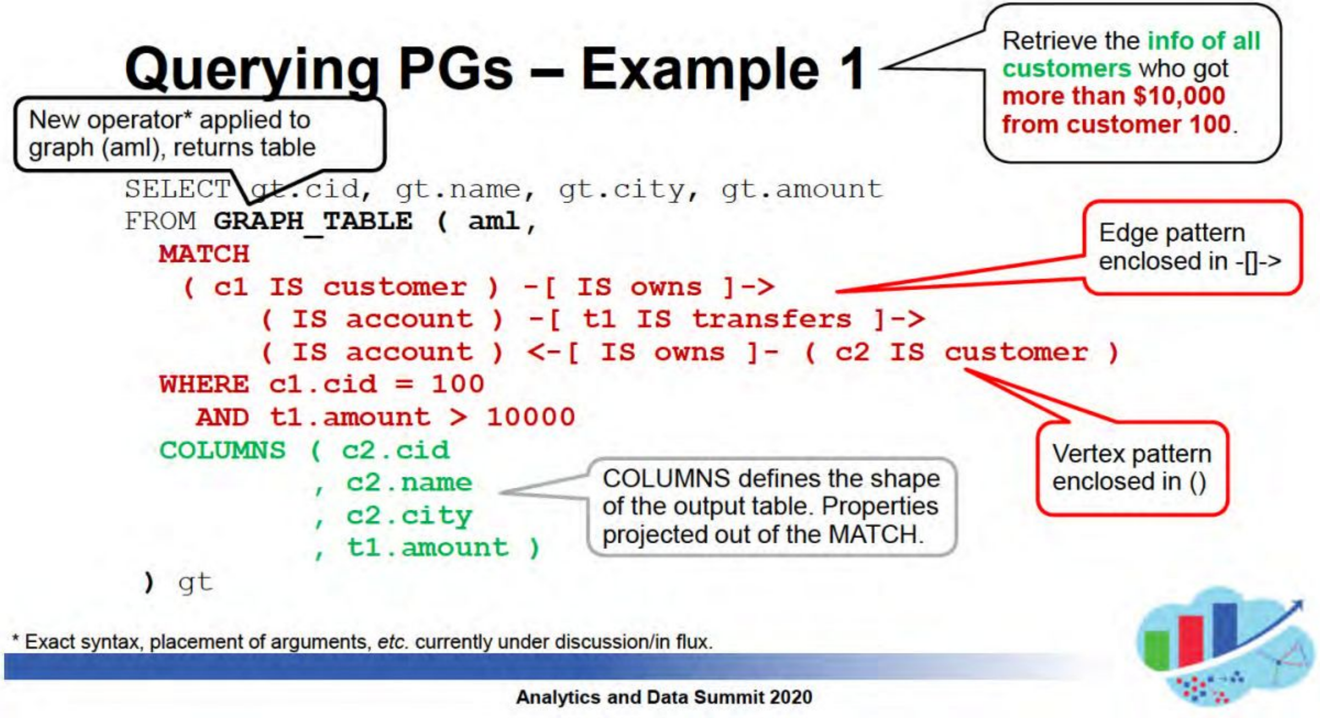
Columns when and amount are exposed as properties. Columns tid, from_id, and to_id are **not**.

Analytics and Data Summit 2020

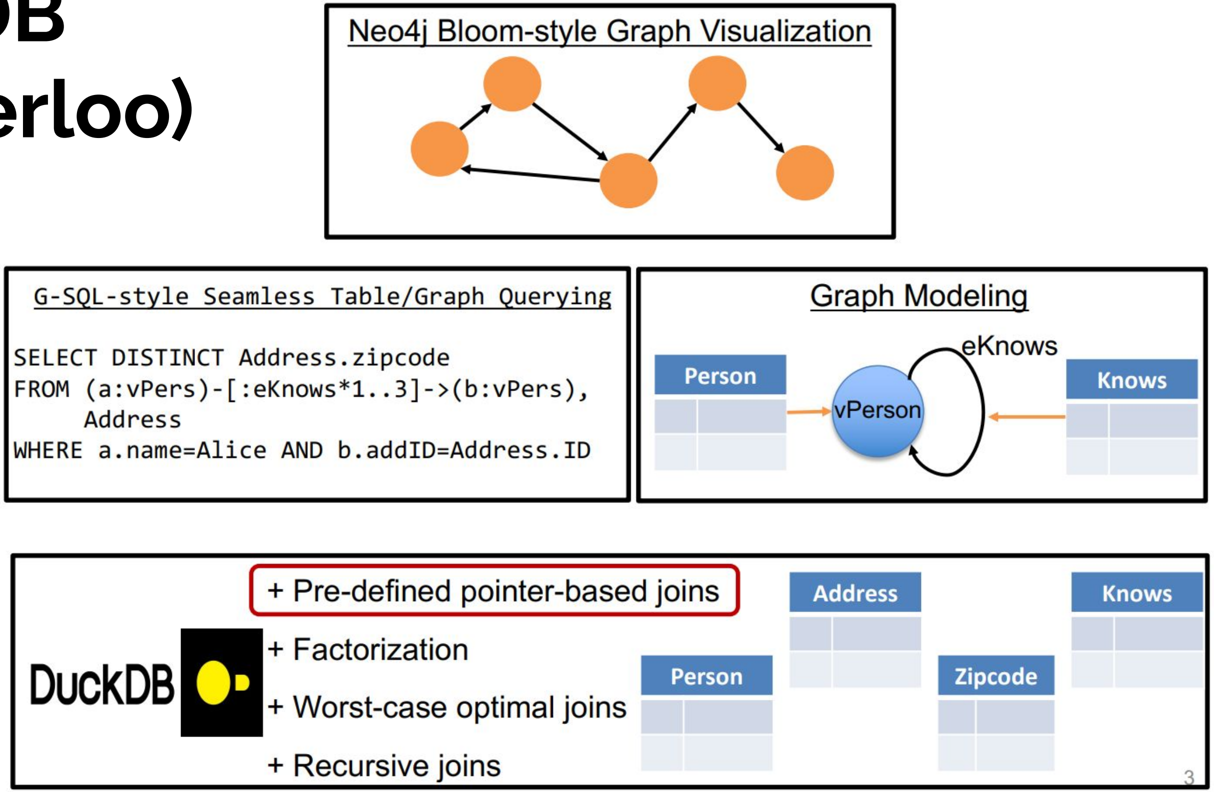
SQL/PGQ: SELECT … FROM GRAPH_TABLE

To be released in 2022

([slides])



GRainDB (UWaterloo)



https://ldbcouncil.org/event/fourteenth-tuc-meeting/attachments/semih-salihoglu-graindb.pdf

Summary

Summary

GDBMSs are a new category of DBMSs focused on graph processing

Many languages -> GQL is much needed

Currently, the best approach may be to build on a (columnar) RDBMS

Further readings on graph processing

Talks:

- GraphflowDB and Modern Query Processing Techniques for GDBMSs (2021) by Semih Salihoglu
- A Survey of Current Property Graph Query Languages (2020) by Peter Boncz
- Graph Processing: A Panoramic View and Some Open Problems (2019) by Tamer Ozsu

Survey papers:

- The future is big graphs: A community view on graph processing systems (2021) by Dagstuhl WG
- Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries (2020) by Maciej Besta et al.
- The ubiquity of large graphs and surprising challenges of graph processing: Extended survey (2020) by Siddhartha Sahu et al.
- Foundations of Modern Graph Query Languages (2017) by Renzo Angles et al.