CWI

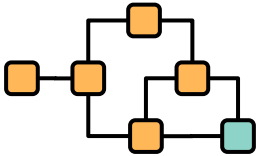
Daniël ten Wolde

The State of DuckPGQ

Peter Boncz (on sabbatical @ MotherDuck)
CWI Database Architectures group

Graph data management

Graph data management



connected data

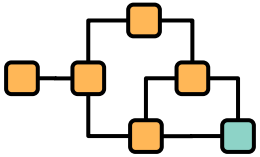
p_id	c_id
1	1
2	2
3	3
...	...

src	dst
1	2
2	3
2	5
...	...

src	dst
4	7
5	7

tables often represent graphs

Graph data management



connected data

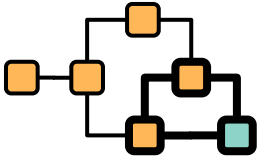
p_id	c_id
1	1
2	
3	
...	

src	dst
1	2
2	3
2	5
...	...

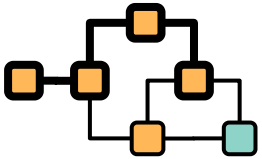
src	dst
4	1
5	1

tables often represent graphs

graph exploration



pattern matching



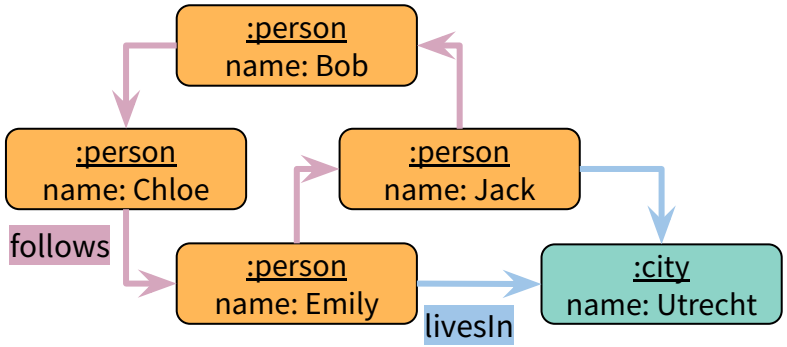
path-finding

```
SELECT count(*)
FROM person
WHERE name LIKE 'E%'
```

relational operators

Storing graphs in SQL

```
CREATE TABLE city (
  id bigint PRIMARY KEY,
  name varchar
);

CREATE TABLE person (
  id bigint PRIMARY KEY,
  name varchar,
  livesIn bigint,
  CONSTRAINT c FOREIGN KEY (livesIn) REFERENCES city (id)
);

CREATE TABLE follows (
  p1id bigint,
  p2id bigint,
  CONSTRAINT p1 FOREIGN KEY (p1id) REFERENCES person (id),
  CONSTRAINT p2 FOREIGN KEY (p2id) REFERENCES person (id)
);
```



Prompt: Count the number of people Bob (in)directly follows who live in the city Utrecht

Prompt: Count the number of people Bob
(in)directly follows who live in the city Utrecht

SQL:1999 query

```
WITH RECURSIVE paths(startNode, endNode, path) AS (
   SELECT p1id AS startNode, p2id AS endNode, ARRAY[p1id, p2id] AS path
     FROM follows JOIN person p1 ON p1.id = follows.p1id WHERE p1.name = 'Bob'
   UNION ALL (
     WITH paths AS (TABLE paths)
        SELECT paths.startNode AS startNode, p2id AS endNode, array_append(path, p2id) AS path
        FROM paths JOIN follows ON paths.endNode = follows.p1id
        WHERE NOT EXISTS (SELECT true FROM paths previous_paths
                          JOIN person p2 ON p2.id = follows.p2id
                          WHERE p2.name = 'Bob' OR follows.p2id = previous_paths.endNode)))
SELECT count(p2.id) AS cp2
FROM person p1
JOIN paths     ON paths.startNode = p1.id
JOIN person p2 ON p2.id = paths.endNode
JOIN city      ON city.id = p2.livesIn AND city.name = 'Utrecht'
```

Prompt: Count the number of people Bob
(in)directly follows who live in the city Utrecht

SQL:1999 query

```
WITH RECURSIVE paths(startNode, endNode, path) AS (
   SELECT p1id AS startNode, p2id AS endNode, ARRAY[p1id, p2id] AS path
     FROM follows JOIN person p1 ON p1.id = follows.p1id WHERE p1.name = 'Bob'
   UNION ALL (
     WITH paths AS (TABLE paths)
        SELECT paths.startNode AS startNode, p2id AS endNode, array_append(path, p2id) AS path
        FROM paths JOIN follows ON paths.endNode = follows.p1id
        WHERE NOT EXISTS (SELECT true FROM paths previous_paths
                          JOIN person p2 ON p2.id = follows.p2id
                          WHERE p2.name = 'Bob' OR follows.p2id = previous_paths.endNode)))
SELECT count(p2.id) AS cp2
FROM person p1
JOIN paths     ON paths.startNode = p1.id
JOIN person p2 ON p2.id = paths.endNode
JOIN city      ON city.id = p2.livesIn AND city.name = 'Utrecht'
```

Prompt: Count the number of people Bob
(in)directly follows who live in the city Utrecht

SQL:1999 query

```
WITH RECURSIVE paths(startNode, endNode, path) AS (
   SELECT p1id AS startNode, p2id AS endNode, ARRAY[p1id, p2id] AS path
     FROM follows JOIN person p1 ON p1.id = follows.p1id WHERE p1.name = 'Bob'
   UNION ALL (
     WITH paths AS (TABLE paths)
        SELECT paths.startNode AS startNode, p2id AS endNode, array_append(path, p2id) AS path
        FROM paths JOIN follows ON paths.endNode = follows.p1id
        WHERE NOT EXISTS (SELECT true FROM paths previous_paths
                          JOIN person p2 ON p2.id = follows.p2id
                          WHERE p2.name = 'Bob' OR follows.p2id = previous_paths.endNode)))
SELECT count(p2.id) AS cp2
FROM person p1
JOIN paths     ON paths.startNode = p1.id
JOIN person p2 ON p2.id = paths.endNode
JOIN city      ON city.id = p2.livesIn AND city.name = 'Utrecht'
```

Graph query languages

Most popular graph system

neo4j

Cypher

Amazon Neptune

SPARQL/Gremlin

JanusGraph

Gremlin

Oracle Labs PGX

PGQL

TigerGraph

GSQL

NebulaGraph

nGQL

SQL/PGQ (Property Graph Queries)

SQL/PGQ

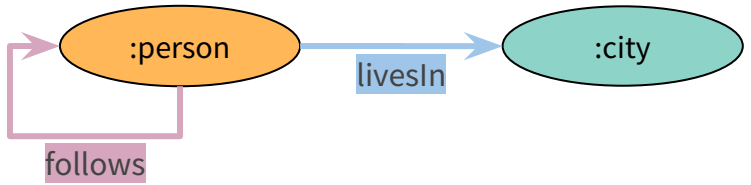
- Extension in the SQL:2023 standard, released in June 2023
- “Property Graphs” defined over existing tables
- Read-only operations for graph queries return “Graph Tables”
    - Path-finding
    - Pattern matching

Tabular schema

```
CREATE TABLE city (
  id bigint PRIMARY KEY,
  name varchar
);

CREATE TABLE person (
  id bigint PRIMARY KEY,
  name varchar,
  livesIn bigint,
  CONSTRAINT c FOREIGN KEY ...
);

CREATE TABLE follows (
  p1id bigint,
  p2id bigint,
  CONSTRAINT p1 FOREIGN KEY ...
  CONSTRAINT p2 FOREIGN KEY ...
);
```

SQL/PGQ graph tables

```
CREATE PROPERTY GRAPH socialNetwork
  VERTEX TABLES (
    city,
    person
  )
  EDGE TABLES (
    livesIn SOURCE person DESTINATION city,
    follows SOURCE person DESTINATION person
  );
```



SQL/PGQ query

Prompt: Count the number of people Bob (in)directly follows who live in the city Utrecht

```
SELECT count(id)
FROM
  GRAPH_TABLE (socialNetwork,
    MATCH (p1:person WHERE p1.name='Bob')-[:follows]->*(p2:person)
          -[:livesIn]->(c:city WHERE c.name='Utrecht')
    COLUMNS (p2.id)
  )
```

SQL/PGQ

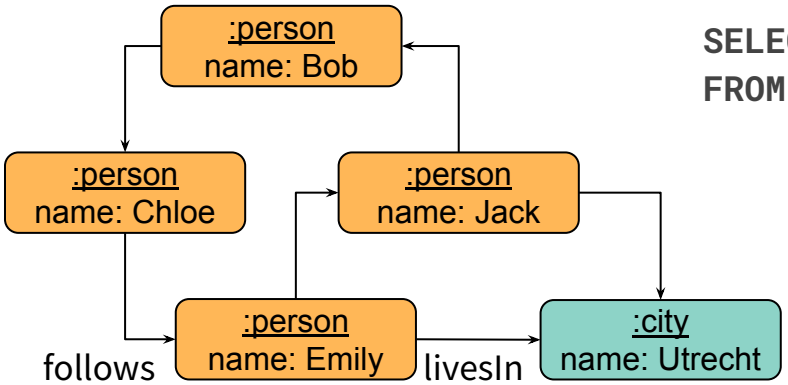
```
SELECT count(id) AS cp2
FROM GRAPH_TABLE (socialNetwork,
  MATCH (p1:person WHERE p1.name='Bob')-[:follows]->*(p2:person)
          -[:livesIn]->(c:city WHERE c.name='Utrecht')
  COLUMNS (p2.id))
```

plain SQL

```
WITH RECURSIVE paths(startNode, endNode, path) AS (
  SELECT p1id AS startNode, p2id AS endNode, ARRAY[p1id, p2id] AS path
    FROM follows JOIN person p1 ON p1.id = follows.p1id WHERE p1.name = 'Bob'
  UNION ALL (
    WITH paths AS (TABLE paths)
      SELECT paths.startNode AS startNode, p2id AS endNode, array_append(path, p2id) AS path
      FROM paths JOIN follows ON paths.endNode = follows.p1id
      WHERE NOT EXISTS (SELECT true FROM paths previous_paths
                        JOIN person p2 ON p2.id = follows.p2id
                        WHERE p2.name = 'Bob' OR follows.p2id = previous_paths.endNode)))
SELECT count(p2.id) AS cp2
FROM person p1
JOIN paths     ON paths.startNode = p1.id
JOIN person p2 ON p2.id = paths.endNode
JOIN city      ON city.id = p2.livesIn AND city.name = 'Utrecht'
```

The SQL/PGQ query is 4× more concise

SQL/PGQ query



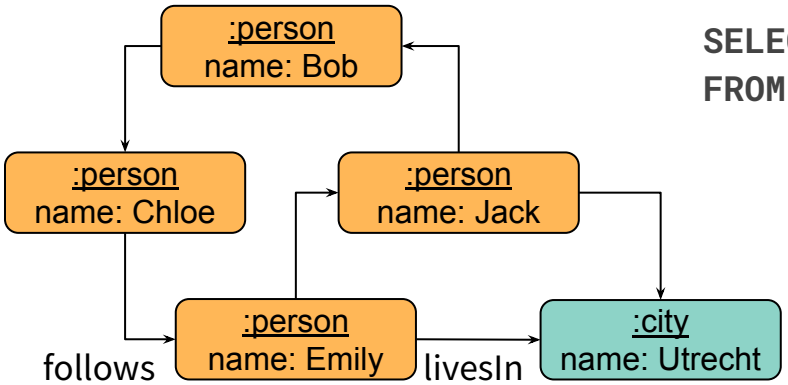

```
SELECT count(id)
FROM GRAPH_TABLE (socialNetwork,
     MATCH (p1:person WHERE p1.name='Bob')-[:follows]->*
                   (p2:person)-[:livesIn]->
                   (c:city WHERE c.name='Utrecht')
     COLUMNS (p2.id))
```

pattern matching

path-finding

relational operators

SQL/PGQ query



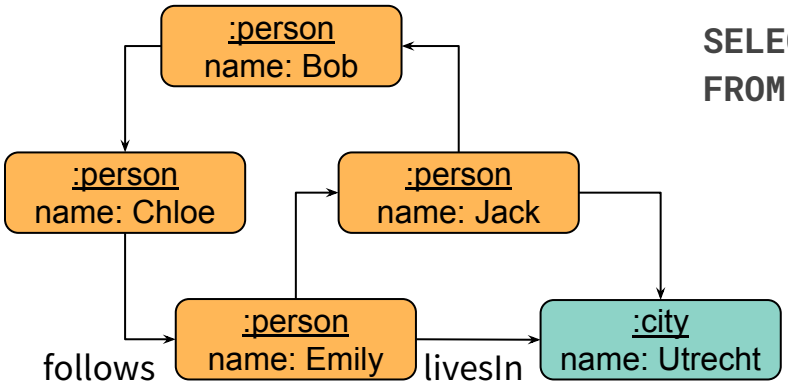

```
SELECT count(id)
FROM GRAPH_TABLE (socialNetwork,
     MATCH (p1:person WHERE p1.name='Bob')-[:follows]->*
                    (p2:person)-[:livesIn]->
                    (c:city WHERE c.name='Utrecht')
     COLUMNS (p2.id))
```

pattern matching

path-finding

relational operators

SQL/PGQ query



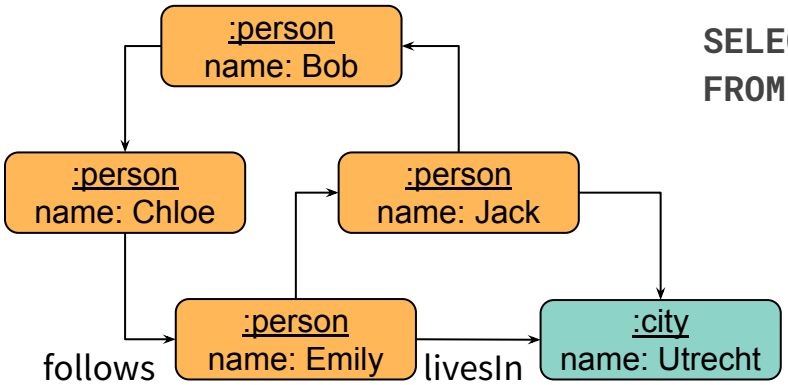

```
SELECT count(id)
FROM GRAPH_TABLE (socialNetwork,
     MATCH (p1:person WHERE p1.name='Bob')-[:follows]->*
                   (p2:person)-[:livesIn]->
                   (c:city WHERE c.name='Utrecht')
     COLUMNS (p2.id))
```

pattern matching

path-finding

relational operators

SQL/PGQ query




```
SELECT count(id)
FROM GRAPH_TABLE (socialNetwork,
     MATCH (p1:person WHERE p1.name='Bob')-[:follows]->*
                    (p2:person)-[:livesIn]->
                    (c:city WHERE c.name='Utrecht')
     COLUMNS (p2.id))
```

pattern matching

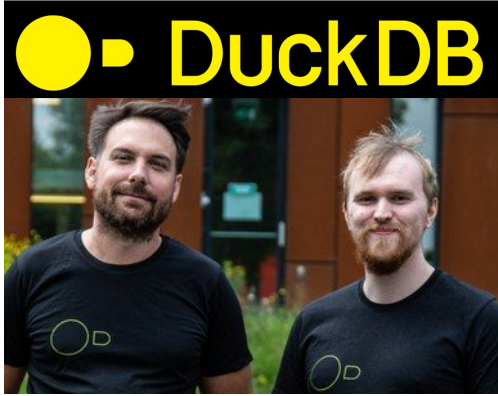
path-finding

relational operators

Implementation of SQL/PGQ in DuckDB

CWI
Centrum Wiskunde & Informatica

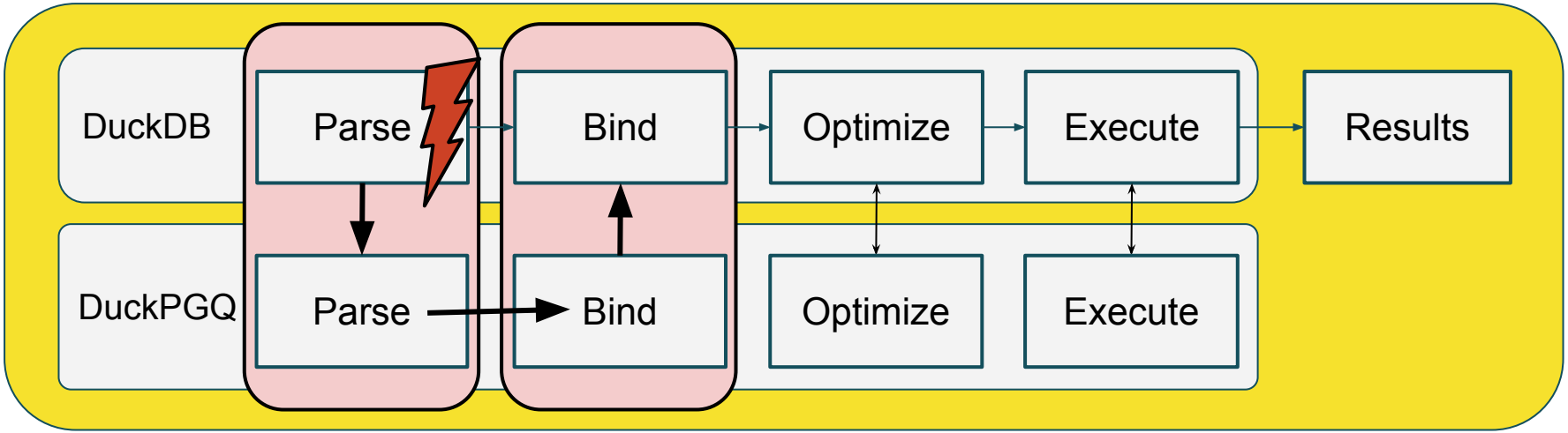
DuckDB: in-process analytics



- Created by Hannes Mühleisen and Mark Raasveldt
- Idea: **analytical SQL system as a linkable library**
- From research on **data systems support for data science:**
  - why don't data scientists use database systems?
    ⇒ make database technology better suited for data science
- Active discord, blog, starting events, traction:
  - 18K github stars, >2M downloads/month (5x increase YoY)
  - DuckDB Labs spin-off (+MotherDuck)

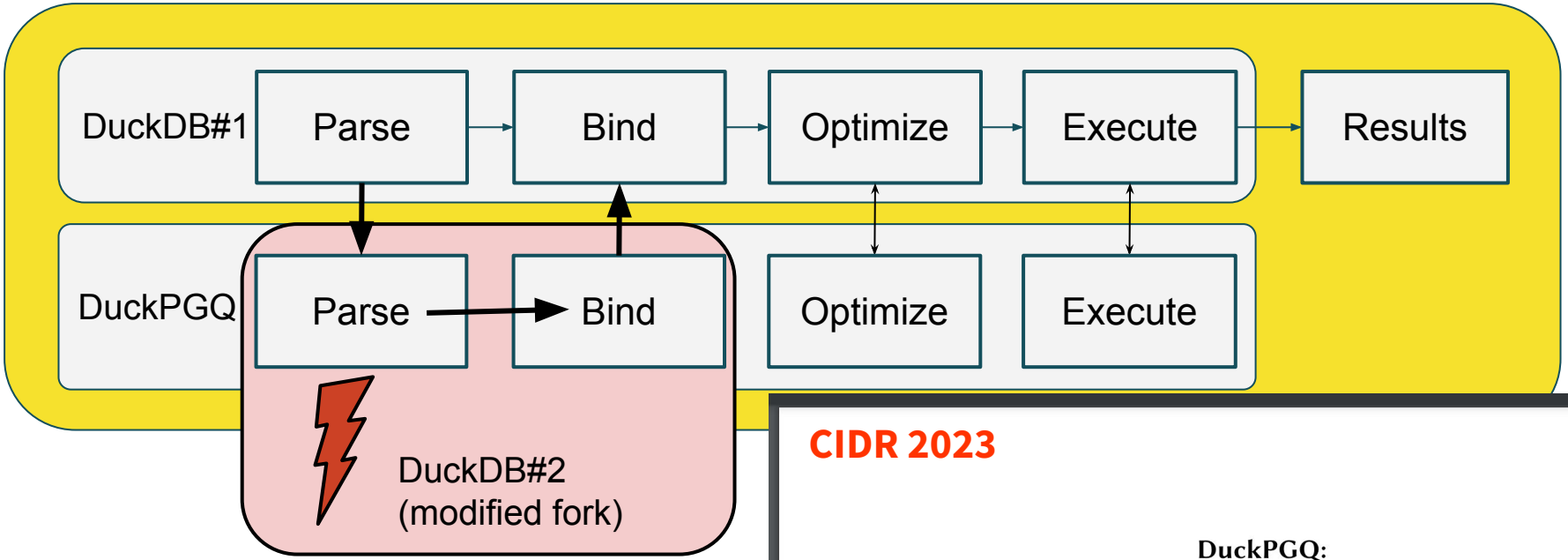
https://duckdb.org/ https://shell.duckdb.org

DuckDB Extension Framework




```
SELECT count(id)
FROM
  GRAPH_TABLE (socialNetwork,
    MATCH (p1:person WHERE p1.name='Bob')-[:follows]->*(p2:person)
            -[:livesIn]->(c:city WHERE c.name='Utrecht')
    COLUMNS (p2.id)
)
```

DuckDB Extension Framework



CIDR 2023

DuckPGQ:
Efficient Property Graph Queries in an analytical RDBMS

Daniel ten Wolde
CWI
The Netherlands
dljtw@cwi.nl

Tavneet Singh
CWI
The Netherlands
tavneet.singh@cwi.nl

Gábor Szárnyas
CWI
The Netherlands
gabor.szarnyas@cwi.nl

Peter Boncz
CWI
The Netherlands
boncz@cwi.nl

ABSTRACT

In the past decade, property graph databases have emerged as a

The upcoming SQL:2023 introduces the **SQL/PGQ** (Pr
Graph Queries) sub-language [8], which allows (1) to define

DuckPGQ: extension module for DuckDB

- Duck **parser extension**
  rewrites queries + UDFs
-

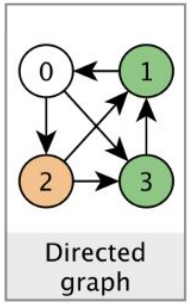
② SQL/PGQ read query

```
SELECT count(gt.p2id) AS p2count
FROM GRAPH_TABLE (sn,
  MATCH (p1:Person WHERE p1.id = 59)
  -[:knows*1..]->(p2:Person)
  -[:likes]->(t:Tag WHERE t.name = 'Oasis')
  COLUMNS (p2.id AS p2id)) gt
```

Rewrite SQL/PGQ query to SQL:1999 queries

③ SQL:1999 query 1 (sketch)

```
SELECT udf_create_edge('kn',
 p1.rowid, p2.rowid, (
  SELECT udf_create_vertex('kn', (
   SELECT count(*) FROM t_pers),
    sub.denseid, sub.outdegree
   FROM (SELECT p.rowid AS denseid,
    count(k.person2id) AS outdegree
   ) sub)))
FROM t_knows
JOIN t_pers p1 ON person1id = p1.id
JOIN t_pers p2 ON person2id = p2.id
```



DuckPGQ: extension module for DuckDB

- Duck **parser extension** rewrites queries + UDFs
- CSR creation **on-the-fly**, exploiting ROWIDs to get dense node-IDs quickly
- Scalar UDFs: multi-core **parallelism** out of the box

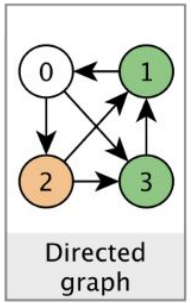
② SQL/PGQ read query

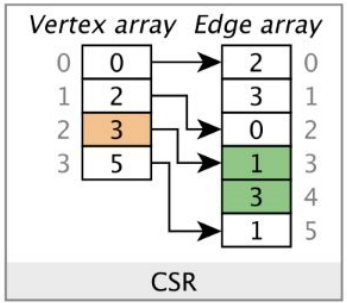
```
SELECT count(gt.p2id) AS p2count
FROM GRAPH_TABLE (sn,
  MATCH (p1:Person WHERE p1.id = 59)
  -[:knows*1..]->(p2:Person)
  -[:likes]->(t:Tag WHERE t.name = 'Oasis')
  COLUMNS (p2.id AS p2id)) gt
```

Rewrite SQL/PGQ query to SQL:1999 queries

③ SQL:1999 query 1 (sketch)

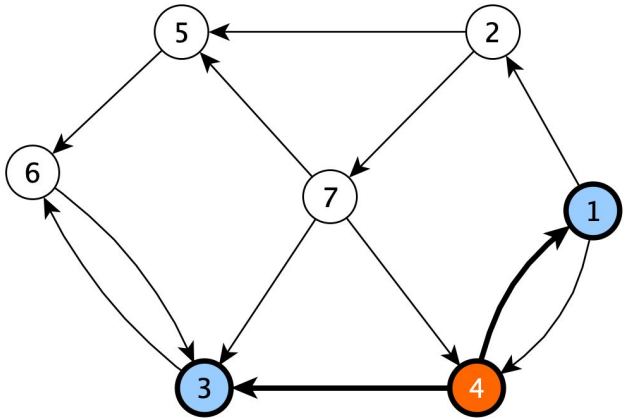
```
SELECT udf_create_edge('kn',
 p1.rowid, b2.rowid, (
  SELECT udf_create_vertex('kn', (
   SELECT count(*) FROM t_pers),
    sub.denseid, sub.outdegree
   FROM (SELECT p.rowid AS denseid,
    count(k.person2id) AS outdegree
   ) sub)))
FROM t_knows
JOIN t_pers p1 ON person1id = p1.id
JOIN t_pers p2 ON person2id = p2.id
```

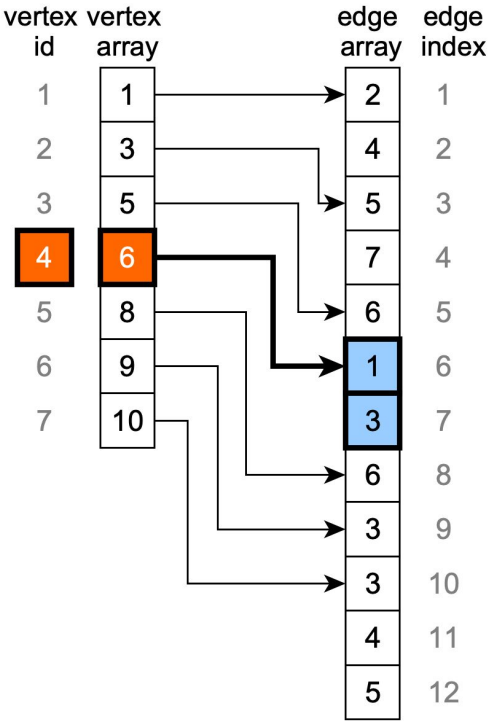




Compressed Sparse Row (CSR) data structure

- On-the-fly creation
- Two scalar UDFs
  - Initialize vertex array
  - Initialize edge array
- Index in the **vertex array** corresponds to the row identifier of the vertex
- Vertex array contains offsets for the **edge arrays**

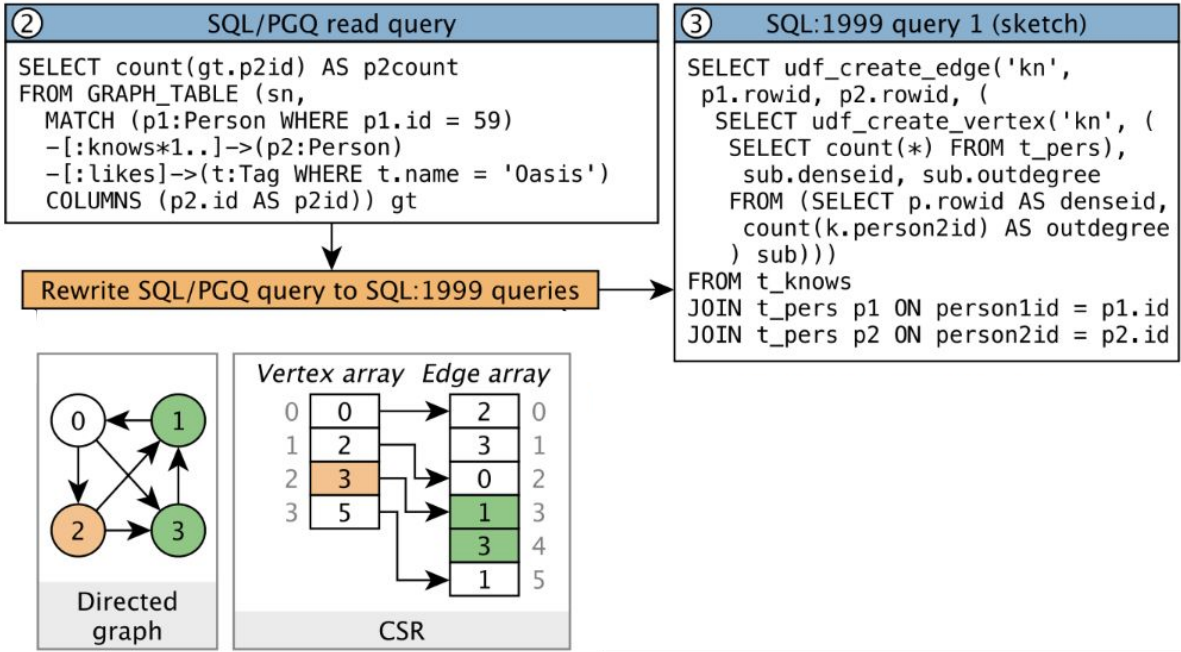




CWI
Centrum Wiskunde & Informatica

DuckPGQ: extension module for DuckDB

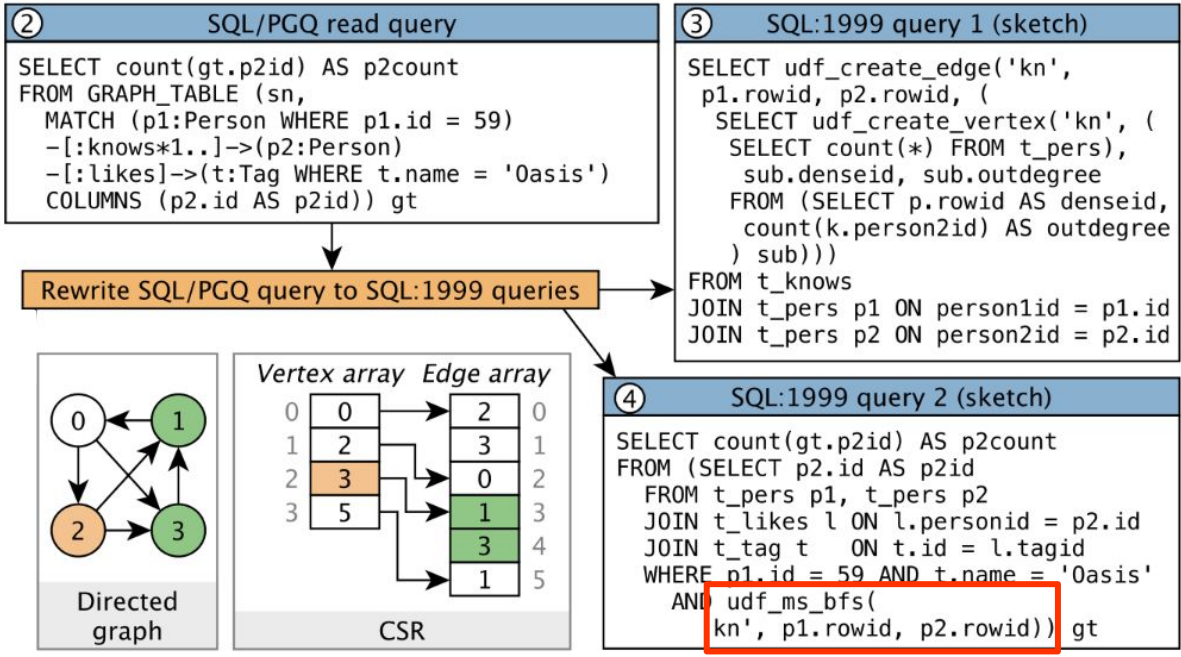
- Duck **parser extension** rewrites queries + UDFs
- CSR creation **on-the-fly**, exploiting ROWIDs to get dense node-IDs quickly
- Scalar UDFs: multi-core **parallelism** out of the box



CWI
Centrum Wiskunde & Informatica

DuckPGQ: extension module for DuckDB

- Duck **parser extension** rewrites queries + UDFs
- CSR creation **on-the-fly**, exploiting ROWIDs to get dense node-IDs quickly
- Scalar UDFs: multi-core **parallelism** out of the box
- **SIMD**-efficient **Multi-Source BFS** (and pathfinding algos)



CWI
Centrum Wiskunde & Informatica

MS-BFS Algorithms

Multiple-Source (MS) graph algorithms

Idea:

- Do many (512 or more) searches at-a-time
  - “**vectorized**”
- Keep state for many (512 or more) searches
- Store state in **SIMD registers**
- Algorithm: sequential access Vertex & Edge arrays
  - Random access in Vertex for destination state
  - but it is shared for 512 searches

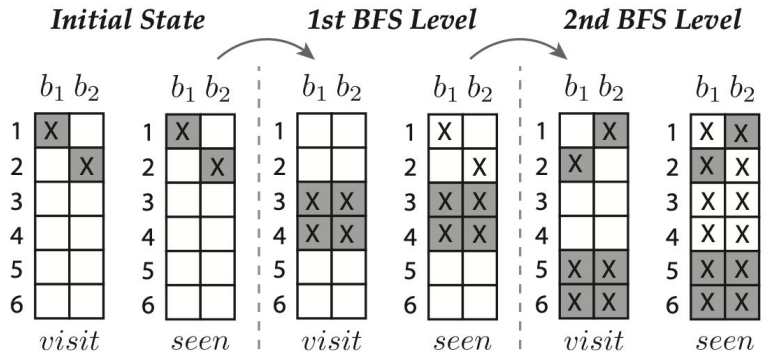


Figure 3: An example showing the steps of MS-BFS when using bit operations. Each row represents the bit field for a vertex, and each column corresponds to one BFS. The symbol X indicates that the value of the bit is 1.

PVLDB 8 (2014)

The More the Merrier:
Efficient Multi-Source Graph Traversal

Manuel Then* then@in.tum.de
Moritz Kaufmann* kaufmanm@in.tum.de
Fernando Chirigati† fchirigati@nyu.edu
Tuan-Anh Hoang-Vu† tuananh@nyu.edu
Kien Pham† kien.pham@nyu.edu
Alfons Kemper* kemper@in.tum.de
Thomas Neumann* neumann@in.tum.de
Huy T. Vo† huy.vo@nyu.edu

* Technische Universität München
† New York University

ABSTRACT

Graph analytics on social networks, Web data, and communication networks has been widely used in a plethora of applications. Many graph analytics algorithms are based on breadth-first search (BFS) graph traversal, which is not only time-consuming for large datasets but also involves much redundant computation when executed multiple times from different start vertices. In this paper, we propose *Multi-Source BFS* (MS-BFS), an algorithm that is designed to run multiple concurrent BFSs over the same graph on a single CPU core while scaling up as the number of cores

have influence on others and, as a consequence, are of great importance to spread information, e.g., for marketing purposes [20].

In a wide range of graph analytics algorithms, including shortest path computation [13], graph centrality calculation [9, 27], and k-hop neighborhood detection [12], *breadth-first search* (BFS)-based *graph traversal* is an elementary building block used to systematically *traverse* a graph, i.e., to visit all reachable vertices and edges of the graph from a given start vertex. Because of the volume and nature of the data, BFS is a computationally expensive operation, lead-

CWI
Centrum Wiskunde & Informatica

DuckPGQ Extensions: weighted path

```
MATCH   ANY SHORTEST PATHS p=(a:Person)-[e:know]->+(b:Person)
COLUMNS (ELEMENT_ID(a) aid, a.name src,
         ELEMENT_ID(b) bid, b.name dst, COST(p), p))
```

aid int64	src varchar	bid int64	dst varchar	COST int32	p int64[]
0	Ana	1	Bo	1	[0, 0, 1]
0	Ana	3	Jo	1	[0, 1, 3]
2	Ed	1	Bo	1	[2, 2, 1]
3	Jo	0	Ana	1	[3, 3, 0]
3	Jo	2	Ed	1	[3, 4, 2]
0	Ana	2	Ed	2	[0, 1, 3, 4, 2]
3	Jo	1	Bo	2	[3, 4, 2, 2, 1]

Note: the green numbers are ELEMENT_IDs of edges

CWI
Centrum Wiskunde & Informatica

DuckPGQ Extensions: label masks

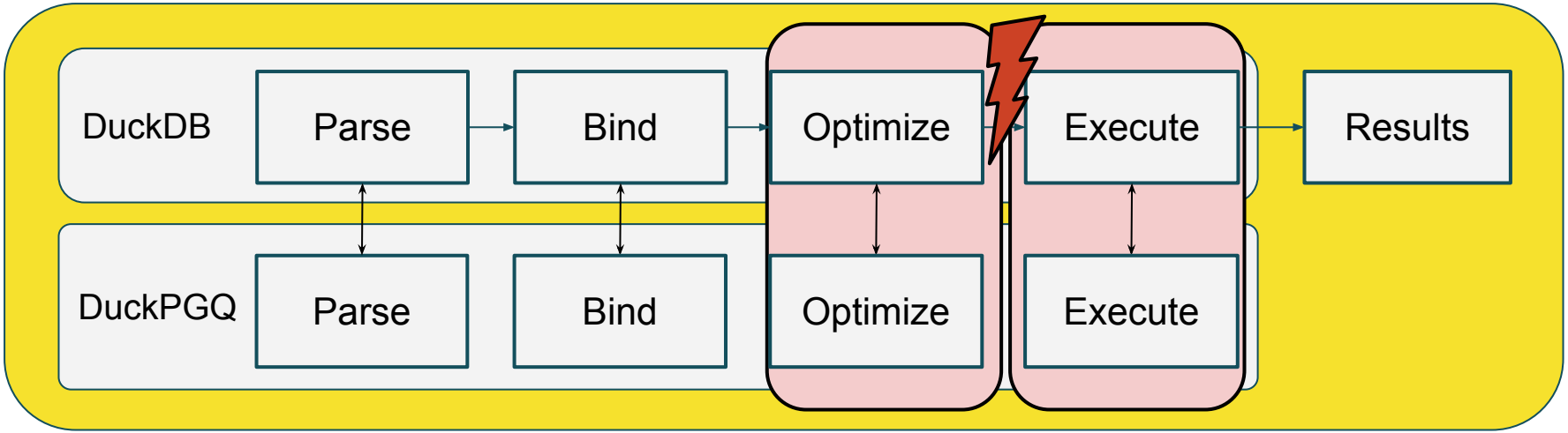
```
CREATE PROPERTY GRAPH pg

VERTEX TABLES (
  College PROPERTIES (id, college)          LABEL College,
  Person  PROPERTIES (id, name, birthDate) LABEL Person
                                               IN msk (Student, TA))
EDGE TABLES (
  know  SOURCE      KEY(src)        REFERENCES Person(id)
        DESTINATION KEY(dst)        REFERENCES Person(id)
        PROPERTIES (createDate, msgCount) LABEL know,

  enrol SOURCE      KEY(studentID) REFERENCES Person(id)
        DESTINATION KEY(collegeID) REFERENCES College(id)
        PROPERTIES (classYear)              LABEL studiesAt );
```

Ongoing work: GNN integration

- Analyze PGQ property graphs in DGL and Pytorch Geometric
- Export DGL and PyTorch Geometric graphs to PGQ property graphs

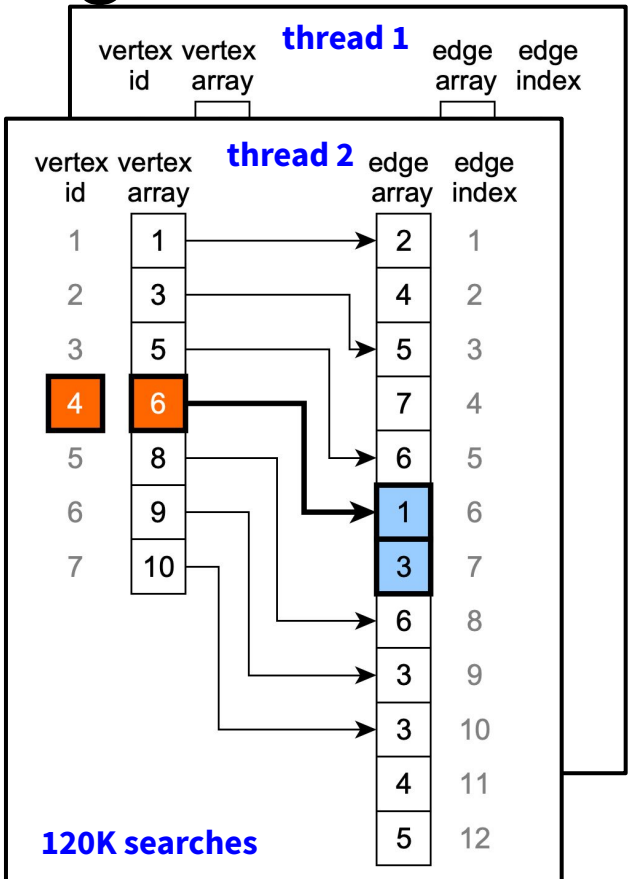
```
import dgl
import torch
import duckdb

# Setup DuckPGQ and collect necessary data
con = duckdb.connect()
csrv, csre, node_features, edge_features = ...
# Initialize a graph object
g = dgl.graph(('csr', (csrv, csre, [])))
# Set the node features, reshaping is necessary
for feature_name, feature in node_features:
    g.ndata[feature_name] = feature.reshape((feature.shape[0], 1))
# Set the edge features, reshaping is necessary
for feature_name, feature in edge_features:
    g.edata[feature_name] = feature.reshape((feature.shape[0], 1))
```

Ongoing work in DuckPGQ



Ongoing work: Parallel Pathfinding

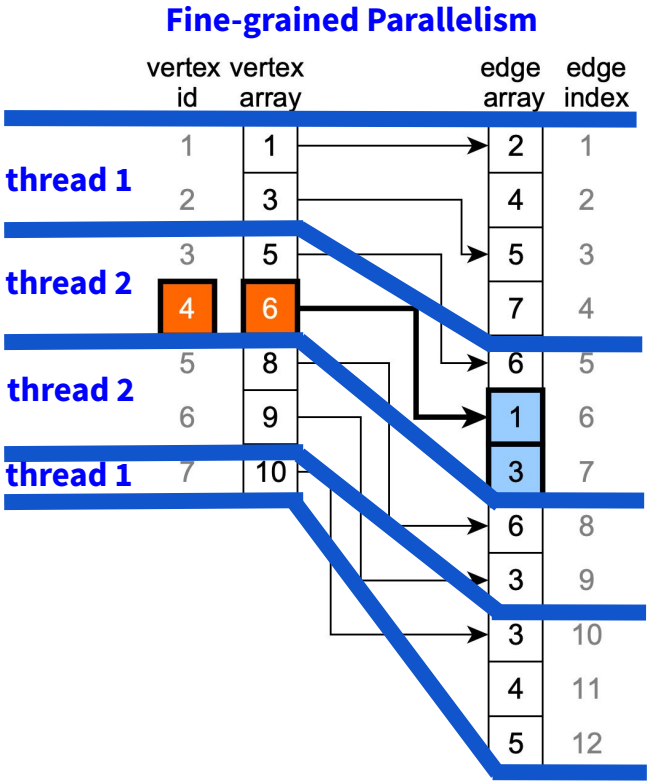
- Scalar function **find_path(src,dst)** has limitations
  - *Morsel-driven* parallelism on [src,dst] table
  - morsel=120K tuples – that's a lot of searches!



Ongoing work: Parallel Pathfinding

- Scalar function **find_path(src,dst)** has limitations
  - *Morsel-driven* parallelism on [src,dst] table
  - morsel=120K tuples – that's a lot of searches!

- New project: DuckDB pathfinding operator
  - Every BFS frontier advance is a DuckDB **event**
  - Threads are **scheduled** to work on vertex-ranges
  - **Source** starts with [src,dst] materialization
    - deduplication of src, dst, [src,dst])
  - **Sink** re-creates the found paths
    - proper order and duplicity

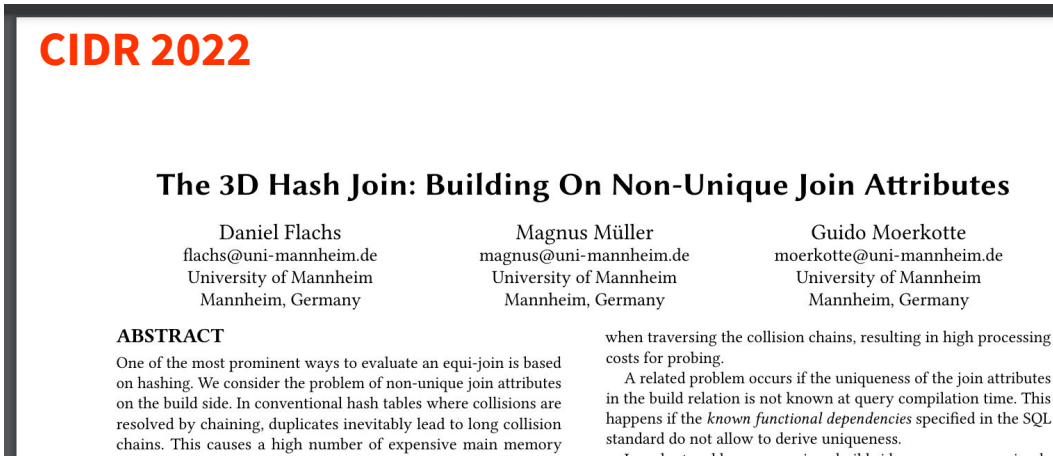


Ongoing work: Factorized Query Processing

- DuckDB for joins uses a bucket-chained hash-table
  - Mixes hash-conflicts with duplicates in a chain
- We changed the hash-table to only have chains for duplicates
  - Linear hashing on the buckets
  - Its' faster on joins!

Ongoing work: Factorized Query Processing

- DuckDB for joins uses a bucket-chained hash-table
  - Mixes hash-conflicts with duplicates in a chain
- We changed the hash-table to only have chains for duplicates
  - Linear hashing on the buckets
  - Its' faster on joins!



CIDR 2022

The 3D Hash Join: Building On Non-Unique Join Attributes

Daniel Flachs
flachs@uni-mannheim.de
University of Mannheim
Mannheim, Germany

Magnus Müller
magnus@uni-mannheim.de
University of Mannheim
Mannheim, Germany

Guido Moerkotte
moerkotte@uni-mannheim.de
University of Mannheim
Mannheim, Germany

ABSTRACT

One of the most prominent ways to evaluate an equi-join is based on hashing. We consider the problem of non-unique join attributes on the build side. In conventional hash tables where collisions are resolved by chaining, duplicates inevitably lead to long collision chains. This causes a high number of expensive main memory

when traversing the collision chains, resulting in high processing costs for probing.

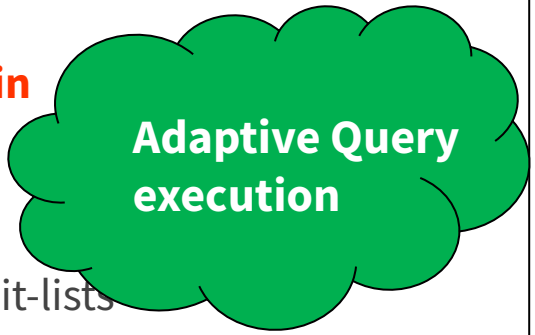
A related problem occurs if the uniqueness of the join attributes in the build relation is not known at query compilation time. This happens if the *known functional dependencies* specified in the SQL standard do not allow to derive uniqueness.

Ongoing work: Factorized Query Processing

- Joins can now return a **hit-list**
  - A **hit-list** points to a hash-table chain ⇒ **factorized n:m join**
- Exploiting these:
  - **Factorized Aggregation**: Embedding (sub-)aggregates in hit-lists
  - **Factorized Joins**: Embedding hit-lists in hit-lists (“graph construction”)
  - **Worst-Case Optimal Joins** (WCOJ): Cyclical joins using hit-list intersection

Ongoing work: Factorized Query Processing

- Joins can now return a **hit-list**
  - A **hit-list** points to a hash-table chain ⇒ **factorized n:m join**
- Exploiting these:
  - **Factorized Aggregation**: Embedding (sub-)aggregates in hit-lists
  - **Factorized Joins**: Embedding hit-lists in hit-lists (“graph construction”)
  - **Worst-Case Optimal Joins** (WCOJ): Cyclical joins using hit-list intersection



Conclusion

- SQL/PGQ (skipped)
- DuckPGQ last year
  - Parser support + scalar UDF MS-BFS pathfinding
  - Extensions: weighted shortest path-ding, flexible labels
- DuckPGQ ongoing work
  - GNN library integrations
  - MS-BFS operator (better parallelism)
  - Factorized query processing

Daniël ten Wolde

DuckPGQ
on GitHub

Try DuckPGQ out in DuckDB v1.0.0

```
> duckdb -unsigned
v1.0.0 1f98600c2c
Enter ".help" for usage hints.
D set custom_extension_repository = 'http://duckpgq.s3.eu-north-1.amazonaws.com';
D force install 'duckpgq';
D load 'duckpgq';
```