A SQL OLAP benchmark based on the LDBC Social Network Benchmark

Nuno Faria

INESCTEC GDC

Graph Data Council Twenty-First TUC Meeting - Boston, Sep 2026

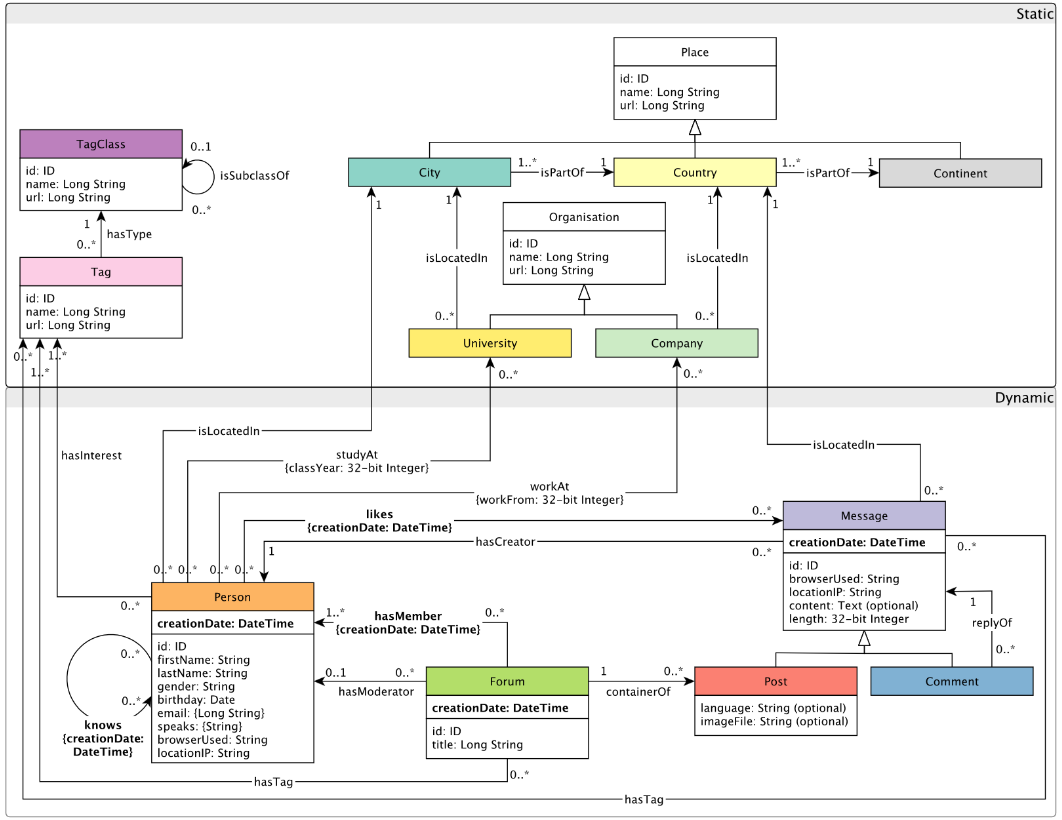
Motivation

- Large sets of data are commonly stored as files (e.g., Parquet) in data lakes, like cloud-based object stores.
- It might not be feasible to pre-load large files to a database to perform analytical workloads.
- Analytical engines that execute directly over these data files (e.g., employing projection and filter pushdown) commonly offer a SQL interface.
- It would be interesting to explore how these systems handle graph-based analytical workloads.

Social Network Benchmark

- Models the data and interactions in a social network. Main entities include persons, forums, and messages.
- Supports two main types of workloads:
  - Interactive
  - **Business Intelligence (BI)**
- BI workload contains 20 analytical queries, each modeling different choke points (e.g., worst-case optimal joins).

Social Network Benchmark schema



Source: The LDBC Social Network Benchmark (version 2.2.4). https://arxiv.org/pdf/2001.02299

Engine Requirements

- SQL analytical engine.
- Can execute directly over Parquet files.
- Ideally supports recursive queries.
  - In the SNB BI benchmark, three queries need recursive support (Q15, Q19, Q20).
- Can be locally deployed.

Recursive Queries

- Allow SQL to execute queries over graph-like structures.
- Implemented with recursive Common Table Expressions (CTEs).
- Turn SQL into a Turing complete language (https://github.com/nuno-faria/tetris-sql)

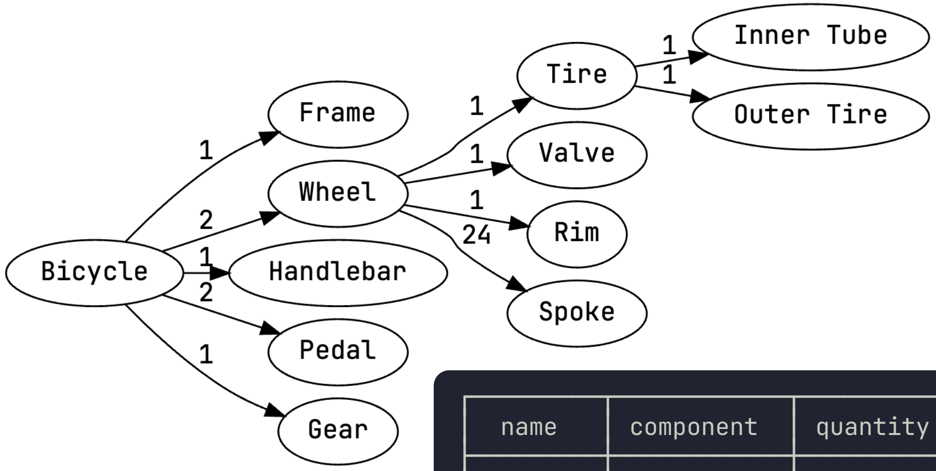
Generator Query

```sql
WITH RECURSIVE t AS
(
    SELECT 1 AS i

    UNION ALL

    SELECT i + 1
    FROM t
    WHERE i < 5
)
SELECT *
FROM t;
```

i
1
2
3
4
5

Recursive Queries

- **Example:** Bill of Materials



name	component	quantity
Bicycle	Frame	1
Bicycle	Wheel	2
Bicycle	Handlebar	1
Bicycle	Pedal	2
Bicycle	Gear	1
Wheel	Tire	1
Wheel	Valve	1
Wheel	Rim	1
Wheel	Spoke	24
Tire	Inner Tube	1
Tire	Outer Tire	1

Cypher

```
MATCH (b:Part {name: 'Bicycle'})
  -[rels:CONTAINS]->+(m:Part)
RETURN m.name AS component,
  reduce(q = 1, r IN rels | q * r.quantity) AS quantity
ORDER BY component;
```

SQL

```
WITH RECURSIVE m AS (
  SELECT component, quantity
  FROM part
  WHERE name = 'Bicycle'
  UNION ALL
  SELECT p.component, m.quantity * p.quantity
  FROM m
  JOIN part p ON p.name = m.component
)
SELECT *
FROM m
ORDER BY component;
```

Survey of Recursive CTE Support

Engine	Supports recursive CTEs?
cedardb	Yes
clickhouse	UNION not supported / Q19 times out
cloudberry	SNB recursive queries crash
databend	UNION not supported / Q19 OOM
datafusion	Cannot be referenced multiple times
doris	Cannot be referenced multiple times
dremio	No
duckdb	Yes
firebolt	No
hive	No
impala	No
oceanbase	Cannot be referenced multiple times
parquet_fdw	Yes
polars	No
presto	No
questdb	No
spark	UNION not supported / OOM / Other limitations
starrocks	Cannot be referenced multiple times / Too slow
trino	Cannot be referenced multiple times / Stack overflow

- The recursive queries in SNB BI (Q15, Q19, Q20) reference recursive CTEs multiple times. Q15 and Q20 also require UNION.

Benchmark Description

- Each system registers the same Parquet files (no data pre-loading).
- 20 SNB BI queries are executed (average of multiple runs).
- Additional queries are created and evaluated to test specific behaviors.

Preliminary Results

SNB BI Queries

query

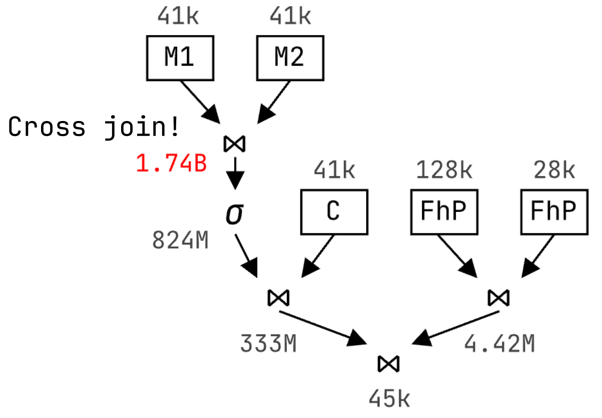
engine	Q01	Q02	Q03	Q04	Q05	Q06	Q07	Q08	Q09	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20
duckdb	0.515	1.07	0.375	0.908	0.681	0.890	0.309	0.238	0.379	0.291	0.024	0.621	1.28	0.460	0.445	0.485	2.91	0.073	4.66	0.982
cedardb	0.555	0.755	1.17	1.23	2.05	1.51	450	1.63	0.800	0.549	0.087	0.712	3.85			0.768		0.194	1.44	0.160
datafusion	0.719	0.638	1.11	3.11	1.10	1.40	0.613	0.601	0.440	0.397	0.076	0.912	1.89	2.92		1.87	5.66	0.105		
polars	1.41	0.438	2.70	3.56	1.21	1.14	0.961	1.76	0.347	0.347	0.042	1.09	1.50	2.87		5.94		0.121		
doris	1.44	1.73	0.468	1.93	2.00	2.49	1.37	0.620	0.595	2.07	0.156	1.36	4.23	2.20		1.72	3.51	0.161		
clickhouse	0.836	1.12	1.40	4.20	2.33	2.74	2.63	1.49	0.479	3.42	0.132	1.32	3.51	1.58		1.71	222	0.366		
starrocks	2.42	1.30	1.22	1.76	1.51	2.42	1.13	1.23	1.09	2.85	0.213	2.69	3.96	2.17		1.65		0.215		
databend	1.29	1.77	4.96	2.08	2.63	3.33	1.49	0.872	0.649	19.0	0.178	2.00	35.2	3.27		6.92	4.47	0.219		
impala	1.40	3.25	2.01	1.55	2.07	3.77	3.59	2.28	0.974	4.08	0.361	1.84	4.68	2.73		3.08	6.84	0.339		
firebolt	1.50	1.77	1.72	6.02	4.36	2.95	5.08	2.25	0.710	3.02	0.149	1.92	6.30	6.23		4.61	10.2	50.2		
dremio	2.14	2.91	2.92	4.92	5.05	4.27	3.38	1.24	1.06	3.29	0.375	2.03	4.63	15.6		3.75	13.6	1.01		
oceanbase	6.73	8.40	3.19	2.08	7.47	6.22	9.17	1.97	1.55	3.83	0.054	3.88	6.77	5.41		13.7		7.37		
parquet_fdw	5.58	8.33	2.09	12.5	45.7	111		2.84	2.07	60.8	0.624	2.61	8.43		1.90	3.59		1.27	10.4	
spark	2.74	5.84	7.72	5.46	5.27	11.5	4.15	2.36	1.92	9.86	0.666	3.08	18.3			6.93	929	1.22		
trino	2.38	4.33	14.8	14.8	25.8	15.6	15.5	5.33	1.12	7.32	0.835	6.65	43.7	28.4		20.5	108	0.741		
questdb	1.69	329	72.3	181	9.33	12.7	5.83	3.86	3.63	9.42	0.447			13.2		9.84	506	1.06		
presto	4.66	16.3	17.0	31.4	15.0	24.4	15.4	7.20	2.21	35.8	2.08	20.2	96.1	45.6		59.9	170	3.23		
cloudberry	22.7	22.1	23.4	32.5	32.2	44.3	33.5	19.4	17.3	33.4	1.08	16.9	34.2			140	51.4	1.57		
hive	12.9	39.5	43.2	116	144	155	121	12.5	38.0	83.9	14.0	18.8	164	216		24.2		6.11		

SF=10, 6 cores, 16GB MEM

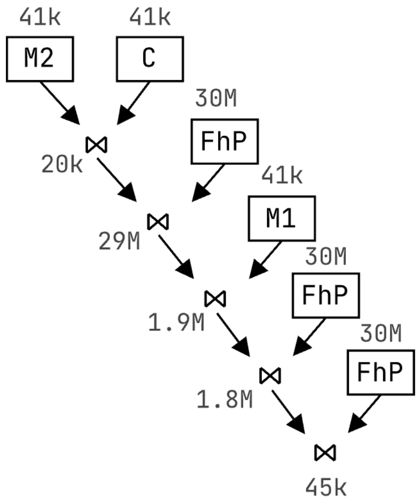
Preliminary Results

- Preliminary conclusions:
  - Q17 is overall the slowest query to execute. Many systems select a sub-optimal join order.

Excerpt of Q17 in ClickHouse



Excerpt of Q17 in DuckDB

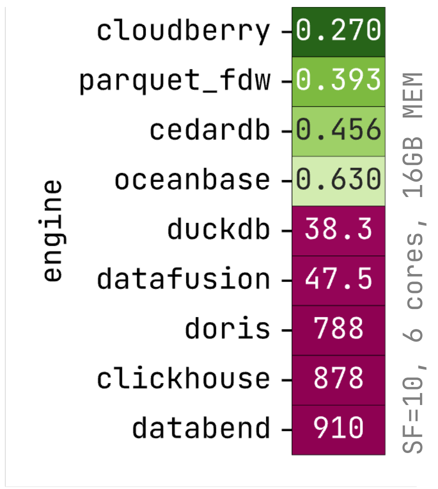


- Bad join ordering is also why Q07 in CedarDB is very expensive.
- DuckDB is the only engine able to execute the full workload.
- There is a large performance disparity between the different systems.

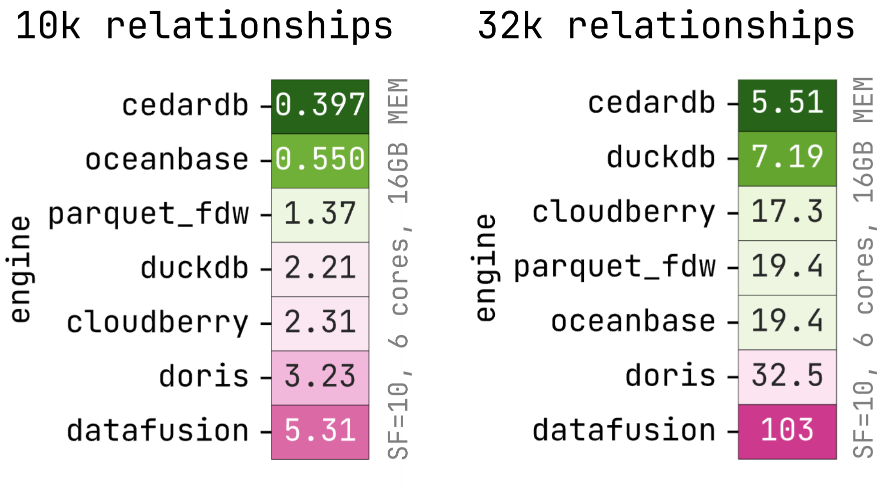
Preliminary Results

- Selected additional queries:

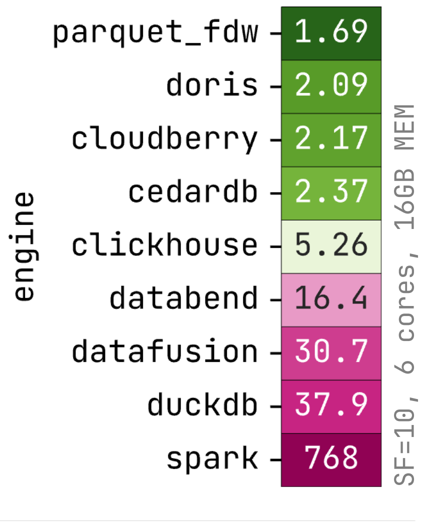
Sequence generator **with a recursive CTE (1M iters)**



Connected components **of a sample of Person_knows_Person relationships**



Single-Source Shortest Paths **(sample of 110k relationships, positive weights)**



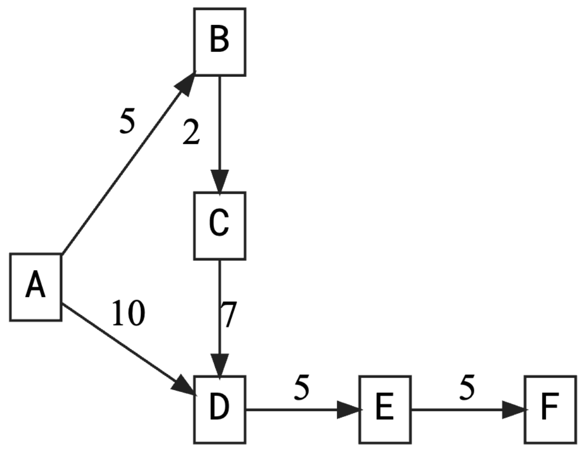
Preliminary Results

- The “single-source shortest paths” query can be improved with the “USING KEY” feature (currently only available in DuckDB).
- Allows recursive queries to access the result (“RECURRING.t”) and update its contents.

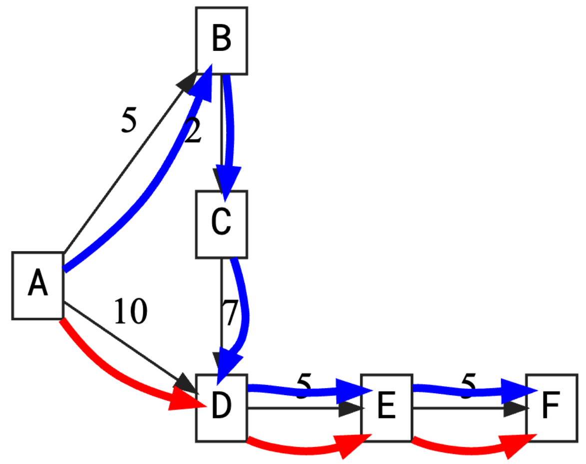
Bamberg, Björn, Denis Hirn, and Torsten Grust. "How DuckDB is USING KEY to Unlock Recursive Query Performance." Companion of the 2025 International Conference on Management of Data. 2025. https://dl.acm.org/doi/pdf/10.1145/3722212.3725107

```
WITH RECURSIVE t USING KEY (k) (
   ...
)
```

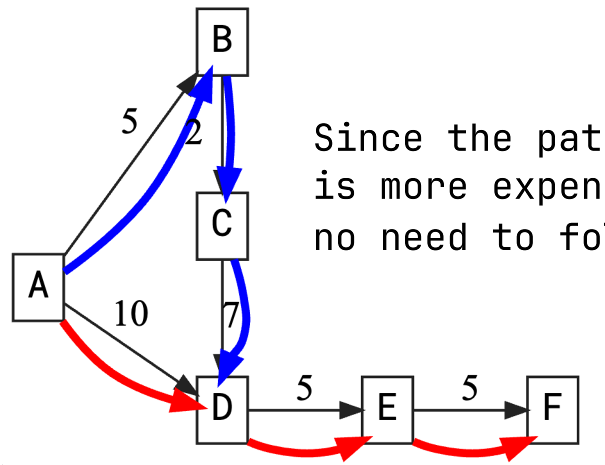
Initial graph



Regular recursive CTE



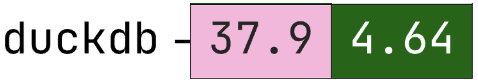
Recursive CTE + USING KEY



Since the path to D from C is more expensive, there is no need to follow it.

Preliminary Results

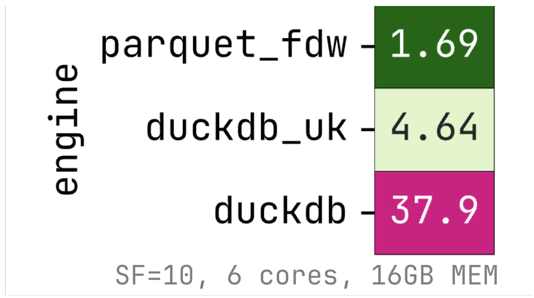
- Single-source shortest paths without (left) and with USING KEY (right) in DuckDB (110k relationships):



SF=10, 6 cores, 16GB MEM

- DuckDB vs DuckDB USING KEY vs parquet_fdw:

110k relationships



130k relationships

SF=10, 6 cores, 16GB MEM

(duckdb runs out of memory)

- Paths computed:
  - 110k relationships:
    - Regular CTE: 194k
    - USING KEY: 9k
  - 130k relationships:
    - Regular CTE: 74M
    - USING KEY: 19k

Summary

- Full recursive CTE support is missing in many analytical systems.
- Correct join ordering is crucial in achieving good results in graph workloads.
- DuckDB shows the overall best performance, CedarDB shows the best recursive CTE performance.
- USING KEY can help improve graph-based algorithms implemented with recursive CTEs

A SQL OLAP benchmark based on the LDBC Social Network Benchmark

Nuno Faria

INESCTEC GDC

Graph Data Council Twenty-First TUC Meeting - Boston, Sep 2026