DuckPGQ: Bringing SQL/PGQ to DuckDB

Daniel ten Wolde
CWI
dljtw@cwi.nl

Gábor Szárnyas
CWI
gabor.szarnyas@cwi.nl

Peter Boncz
CWI
boncz@cwi.nl

ABSTRACT

We demonstrate the most important new feature of SQL:2023, namely SQL/PGQ, which eases querying graphs using SQL by introducing new syntax for pattern matching and (shortest) path-finding. We show how support for SQL/PGQ can be integrated into an RDBMS, specifically in the DuckDB system, using an extension module called DuckPGQ. As such, we also demonstrate the use of the DuckDB extensibility mechanism, which allows us to add new functions, data types, operators, optimizer rules, storage systems, and even parsers to DuckDB. We also describe the new data structures and algorithms that the DuckPGQ module is based on, and how they are injected into SQL plans.

While the demonstrated DuckPGQ extension module is lean and efficient, we sketch a roadmap to (i) improve its performance through new algorithms (factorized and WCOJ) and better parallelism and (ii) extend its functionality to scenarios beyond SQL, e.g., building and analyzing Graph Neural Networks.

PVLDB Reference Format:
Daniel ten Wolde, Gábor Szárnyas, and Peter Boncz. DuckPGQ: Bringing SQL/PGQ to DuckDB. PVLDB, 16(12): 4034 - 4037, 2023.
doi:10.14778/3611540.3611614

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/cwida/duckpgq-extension.

1 SQL/PGQ

SQL/PGQ (Property Graph Queries) is part of the recently released standard SQL:2023, and enables SQL users "to see graphs in tables". As an intuition, the standard mathematical notation $G(V, E)$ of graphs can be used in relational database systems by seeing V and E as respectively a vertex and edge table.

A vertex table is a normal SQL table that represents each vertex as a tuple and needs to have a unique key – though having an explicit SQL constraint for that is not enforced in SQL/PGQ. Similarly, an edge table consists of tuples that represent edges. It must contain two referencing columns (or column combinations, in case of multi-column keys), i.e., foreign keys (FKs); where these two respectively point to the source and destination vertex of the edge. Finally, the notion of *properties*, well-known from the *property graph* data model (e.g., used in systems [5] such as Neo4j, TigerGraph, Dgraph, JanusGraph, Oracle PGX, and AWS Neptune) is represented by the non-key columns in the edge and vertex tables.

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611614

SQL/PGQ is the first property graph query (sub-)language supported by a standards organization (ISO). The graph database systems we just mentioned all use a different graph query language [2]: respectively Cypher, GSQL, GraphQL, Gremlin, PGQL and SPARQL – the latter admittedly also a standard (recommendation) from W3C, but not for the property graph data model. This fragmentation of query languages is not conducive to growing graph database adoption, and therefore it is fortunate that the industry behind these systems started to put efforts into standardization. The first result of this effort and topic of this demo, SQL/PGQ, adopts ideas from previous graph query languages: e.g., the pattern matching syntax is inspired by Neo4j's Cypher, and the path-finding expressions were influenced by the PGQL language in Oracle PGX. Another influence has been the G-CORE language [3] proposed by a consortium of academia and industry brought together by the Linked Data Benchmark Council (LDBC). LDBC is an organization founded in 2013, initially focused on benchmarking but is now also active in standardization. It has an ongoing liaison with the ISO working group for SQL, which allows LDBC members access to ISO design documents[1]. ISO has almost finalized GQL [10], a native graph query language that will eventually be able to not only return tabular results but also graphs; and with LDBC is designing a specification for property graph schemas (early work is in [4]).

```
CREATE PROPERTY GRAPH pg
VERTEX TABLES(
 Person     PROPERTIES(id, firstName) LABEL Person,
 University PROPERTIES(id, name)      LABEL University,
 Message    PROPERTIES(messageId, content)
            LABEL Message IN MessageType(Post, Comment))
EDGE TABLES(
 Person_knows_Person
  SOURCE KEY(person1Id) REFERENCES Person(id)
  DESTINATION KEY(person2Id) REFERENCES Person(id)
  PROPERTIES(creationDate, interactionCount)
  LABEL know,
 Person_likes_Message
  SOURCE KEY(personId) REFERENCES Person(id)
  DESTINATION KEY(messageId) REFERENCES Message(id)
  PROPERTIES(creationDate)
  LABEL likes,
 Person_studyAt_University
  SOURCE KEY(personId) REFERENCES Person(id)
  DESTINATION KEY(universityId) REFERENCES University(id)
  PROPERTIES(classYear)
  LABEL studyAt);
```

Listing 1: SQL/PGQ query to create a property graph

The above statement defines a property graph storing a social graph of persons who know each other, like each other's messages, and study at universities.

After creating this property graph (with name `pg`), it can be used in queries. At the heart of SQL/PGQ is the new `MATCH` clause [7],

[1] One can become a personal member of LDBC for free at ldbcouncil.org.
Gábor Szárnyas in the past years helped manage LDBC as a member of its steering committee, and Peter Boncz is co-founder and current chair of LDBC.

which embeds Cypher-like graph pattern queries, where vertex patterns are denoted with round parentheses and edge patterns with square brackets. Restrictions on labels are denoted `v:label`, where `v` is a (tuple) variable to bind and `label` is a label restriction. It is possible to add additional `WHERE` filters inside both vertex and edge patterns. The edges use "ASCII art" notation to specify conditions for matching edges, i.e., right- (`-[]->`), left- (`<-[]-`), left-right- (`<-[]->`), and any-directed (`-[]-`) edge patterns.

Our first example query doing pattern-matching asks for years and university names where Bob studied:

```
SELECT study.classYear, study.name
FROM GRAPH_TABLE (pg,
     MATCH (a:Person WHERE a.firstName = 'Bob')
            -[s:studyAt]->(u:University)
     COLUMNS (s.classYear, u.name)) study;
```

Listing 2: Q1: basic pattern matching

The result of matching is an imaginary *binding table*, where each column is a variable (`a`, `s`, `u`) and each row a binding that matches all constraints in the pattern. But these variables are tuple variables, not scalar values. Hence, the role of the `COLUMNS` clause is to turn tuple variables into scalar values, by selecting properties, e.g., `s.classYear` selects property `classYear` of `studyAt` edges matched on variable `s`. The result of this all is a table `study(classYear, name)`, returned by the SQL/PGQ table function `GRAPH_TABLE` in the `FROM` clause of SQL; and hence it can be further processed with SQL (filtered, joined with other tables, aggregated, etc.). SQL/PGQ also supports both bounded path queries, where a lower and upper amount of hops between two nodes is specified (e.g., `-[]->{1,4}`) as well as unbounded path queries, where no upper limit is defined (`-[]->*` or `-[]->+`).

```
SELECT friends.p2_firstName
FROM GRAPH_TABLE (pg,
     MATCH (a:Person WHERE a.firstName = 'Bob')
            -[s:know]->* (p2:Person)
     COLUMNS (p2.firstName)) friends;
```

Listing 3: Q2: basic path-finding (reachability)

This second example query is about reachability: we are looking for all persons `p2` Bob can reach over `know` edges, but we do not return the specific paths. The query could have asked to return a path by adding `SHORTEST p =` just after `MATCH`, turning this into a path-finding query that would return one (any) shortest path from Bob to each reachable person `p2`.

2 DUCKPGQ: ADDING SQL/PGQ TO DUCKDB

DuckDB is an extensible state-of-the-art single-node analytical database system [23]; featuring columnar storage with MinMax predicate pushdown [1], lightweight compression [27] on all datatypes including strings [6] and doubles [18], morsel-driven parallel multi-core execution [16], complete query decorrelation [19], full support for window functions [17], dynamic programming-optimized join order enumeration [20], cardinality estimates based on samples and Hyperloglogs [11], a compressed-vector respresentation (shared with Velox [22]), out-of-core algorithms for join, aggregation and sort [15] as well as adaptive mapping of semi-structured data in CSV [8] and JSON[2] to DuckDB's non-first-normal-form data model that shreds column values that are lists or structs or nested combinations thereof into internal columns.

[2]Shredding deeply nested JSON, one vector at a time: bit.ly/duckdb-json-laurens-kuiper

The life of a query in DuckDB passes through the following stages: (i) parsing SQL into a syntax tree, (ii) transforming the syntax trees into a parsed statement representation, (iii) binding of table names (using catalog lookups) and typing of expressions as well as performing all semantic checks and any needed error raising (iv) generating a logical relational algebra plan and (v) optimizing this query plan and generating a physical query operator tree, followed finally by (vi) query execution – using push-based morsel-driven vectorized execution; where operators in a pipeline push data in chunks (using a vector size of 2048 tuples) from a source operator (e.g., a table scan) through streaming operators (such as a hash-join probe or filter) upwards to the top of the pipeline where results are materialized in a sink operator (e.g., a hash-table build).

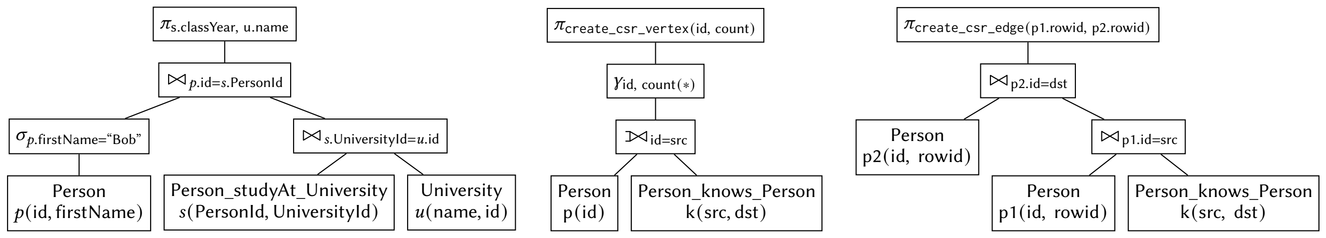
The original creators of DuckDB, Hannes Mühleisen and Mark Raasveldt, and CWI have moved the core system code to an open-source foundation (duckdb.org) and also created a spin-off company (DuckDB Labs) that aims to grow the adoption of the system under its open-source charter. Their efforts have resulted in sharp growth in the adoption of DuckDB. It is the first system in its own class of *embeddable analytics*: it is used as a library and runs inside, e.g., the R or Python interpreter, such that data scientists working on notebooks using popular packages such as dplyr and NumPy/pandas (and PyTorch and TensorFlow) can get *zero-copy* SQL access to dataframes. While data science is the home turf of DuckDB adoption, its low footprint and embeddable nature make it suitable for many (new) use cases, including mobile applications, edge computing, in-browser computing, embedding in storage devices (computational storage); but of course also cloud computing. The DuckDB founders have also co-founded a startup for cloud-based DuckDB computing (MotherDuck).

DuckDB is purely written in C++ with no software dependencies, nor explicit usage of platform-specific code such as assembly or intrinsics in order to keep the system portable. These characteristics also allow it to be compiled to WebAssembly [14] and therefore run inside any modern web browser (demo of that: shell.duckdb.org).

DuckDB provides the possibility to create C++ *extension modules* that can add new scalar and aggregation functions, data types, relational operators, optimizer rules that put these to use, and even new transactional storage systems or query parsers to DuckDB. The parser extension feature of DuckDB registers a separate parser, which DuckDB invokes after syntax error in its SQL parser, which is a C++ port of the PostgreSQL parser.

DuckPGQ is the result of a research project at CWI into efficient graph analytics systems design. The support for SQL/PGQ – the topic of this demonstration – is on the one hand a central feature, but only the stepping stone in this project, which aims beyond the query language support also for fluent interaction with dataframe libraries (dplyr, NumPy/pandas and Arrow), but also tools for machine learning and specifically training, analyzing and scoring Graph Neural Networks [13] and graph visualization (e.g., NetworkX).

Supporting SQL/PGQ in DuckDB exercises its extensibility API in various ways. The DuckDB parser extension API was originally intended to allow extension modules to introduce new statement types, but in the case of DuckPGQ, we register a complete SQL



$\pi_{s.\text{classYear, u.name}}$
$\bowtie_{p.\text{id}=s.\text{PersonId}}$
$\sigma_{p.\text{firstName}=\text{“Bob”}}$
$\bowtie_{s.\text{UniversityId}=u.\text{id}}$
Person p(id, firstName)
Person_studyAt_University s(PersonId, UniversityId)
University u(name, id)
$\pi_{\text{create\_csr\_vertex(id, count)}}$
$\gamma_{\text{id, count}(*)}$
$⟖_{\text{id=src}}$
Person p(id)
Person_knows_Person k(src, dst)
$\pi_{\text{create\_csr\_edge(p1.rowid, p2.rowid)}}$
$\bowtie_{\text{p2.id=dst}}$
Person p2(id, rowid)
$\bowtie_{\text{p1.id=src}}$
Person p1(id, rowid)
Person_knows_Person k(src, dst)

(a) Relational algebra tree for Q1 (defined in Listing 2) **(b) `create_csr_vertex`** **(c) `create_csr_edge`**

Figure 1: Relational algebra trees for expressing Q1 and the user-defined functions `create_csr_vertex` and `create_csr_edge`

parser built from a fork of DuckDB; that is, it parses a superset of SQL with the addition being PGQ.

The initial version of DuckPGQ that we will demo in fact only affects the first three stages in the life of a query: parsing, transforming, and binding, as it maps the new SQL/PGQ functionality into a traditional logical SQL query plan that mainline DuckDB can handle, with the help of some scalar User Defined Functions (UDFs) implemented in C++ that the module provides. This SQL-rewriting approach is sufficient for translating all SQL/PGQ functionality. For example, the first example query gets converted to the plan depicted in Figure 1a. This plan contains two equi-joins on the PK-FK relationships defined during the creation of the edge table for this property graph (arguably the join order is not optimal here, but this way it is shorter so let us consider that a logical plan).

For path-finding and reachability, we decided not to map to `WITH RECURSIVE` SQL query plans, for performance reasons but instead introduced scalar UDFs that create a Compressed Sparse Row (CSR) data structure on the fly and which is used for path-finding. The CSR data structure consists of a vertex array and an edge array: the latter contains the vertex destination-positions of all edges. It consecutively holds the group of destination vertex-positions of all edges starting at one source vertex and holds these groups consecutively in the storage order of the vertex array. The vertex array contains the position in the edge array corresponding to its first outgoing edge. We create the CSR using two scalar UDFs in DuckDB that fill either CSR array with the result of a subquery we generate. For `create_csr_vertex()` the subquery delivers dense vertex IDs and their edge count (a `count` aggregate over an outer join between vertex and edge table) as shown in Figure 1b. For `create_csr_edge()` the subquery delivers for all edges, the dense vertex IDs for its source and destination (a double join between edge and vertex tables, on the source and destination keys, respectively), see Figure 1c. Rather than generating dense vertex IDs using a `DENSE_RANK()` window function, we exploit the fact that DuckDB tables already have `ROWIDs` that are semi-dense integers; wasting a bit of space on possible holes left by non-checkpointed deletes. The fact that the CSR-constructed subquery, that we generate under the hood, is implemented with simple scalar UDFs – which a vectorized engine executes just as fast as builtin expressions – makes DuckPGQ profit from multi-core parallelism in DuckDB out-of-the-box.

Our motivation to create a CSR on the fly for each path-finding predicate is that the alternative, mapping into `WITH RECURSIVE` queries, would typically end up using hash-joins and specifically a hash-table lookup for each traversed edge. We think that the investment in creating a CSR data structure that allows positional lookup is typically worthwhile, due to the large number of lookups required for path-finding. A second reason is that such a data structure provides a good basis for Worst-Case Optimal Join (WCOJ) algorithms [12] and factorized query execution. A third reason is that CSR data structures are not only useful for query processing: they also form the backbone of GNN libraries such as DGL and PyTorch-geometric [21] and when linked into a Python process, DuckPGQ eventually aims to provide zero-copy access to ingest graphs into these machine learning tools, as well as the possibility to analyze the trained graphs using SQL/PGQ.

Most of this is in the future roadmap and has not yet been implemented yet. What has been implemented, and will be demonstrated, are the Multi-Source BFS algorithms (MS-BFS [26], which can be used for computing reachability, simple path-finding, and shortest path-finding (with an additional weight column, adjacent to the CSR edge array). If paths need to be returned we keep yet another adjacent array of edge `ROWIDS` – DuckPGQ returns paths as DuckDB lists of alternating vertex and edge `ROWIDs` that point back into the vertex and edge tables. SQL/PGQ introduces the `ELEMENT_ID(e)` function that returns a numeric internal identifier for elements (vertex or edge variables), and DuckPGQ implements these simply as `ROWIDS`.

The MS-BFS algorithm is a good match for DuckPGQ because of its vectorized nature: its performance advantage comes from the bulk sequential access through a CSR data structure, and the fact that MS-BFS executes many path searches simultaneously, exploiting the power of SIMD instructions: for keeping track of which vertexes have been visited a single bit suffices, such that an AVX512 register can keep track of 512 searches. Note that this does not require use of assembly or intrinsics, well-coded MS-BFS algorithms can be auto-vectorized efficiently by C++ compilers. The benefits of MS-BFS stem from the need to execute fewer CPU instructions in total, as instructions perform work for multiple active searches, as well as thanks to reduces memory access.

We implemented path-finding using simple scalar UDFs that as input get a vector of source vertexes and a vector of destination vertexes, such that each call to such a UDF is tasked to handle 2048 path-finding searches. This fits MS-BFS, as it needs at least hundreds of searches to exploit SIMD. Like with CSR creation, UDFs again provide out-of-the-box parallelism for path-finding making DuckPGQ outperform all graph database systems we tested [25].

However, this simple way of implementing MS-BFS requires queries that generate thousands of searches, and not all queries do so. Hence we believe we will eventually implement MS-BFS with a different mechanism (possible useful DuckDB extensibility mechanisms are: creating an input-output table function, as used in `LATERAL JOIN` queries; or creating a whole new relational operator). However, the horizontal parallelism that would befit MS-BFS, where each breadth-first step would horizontally partition the vertexes in the CSR over multiple threads, would need to be elegantly and efficiently integrated with the DuckDB parallelism model; which provides another research item on our roadmap.

Beyond SQL/PGQ. The SQL/PGQ `CREATE PROPERTY GRAPH` statement gives all tuples in a vertex or edge table the same label or set of labels. We found this mechanism alone too rigid and implemented an extension that allows for flexible assignment of labels, that is e.g., capable of expressing inheritance, see vertex table `Message` in Listing 1. The label `Message` is a global label that applies to all entries in the `Message` table. In DuckPGQ, a so-called *discriminator* column, here `MessageType`, contains integers – interpreted as bitmaps. Bit x is then set iff the element is part of the subset listed as the xth element in the list following the discriminator. So in this example, the value 1 in `Message.MessageType` means the message has label `Post` and a value 2 label `Comment`. All vertexes in this table have the global label `Message`. A value of 3 would mean a vertex has all three labels.

Currently, the SQL/PGQ specification only defines how to perform shortest path-finding, while cheapest path-finding is marked as a "language opportunity" [7]. DuckPGQ supports cheapest path-finding using the keyword `COST` followed by an expression defining edge weights. DuckDB uses a SIMD-friendly variant of MS-BFS, Multi-Source Bellman-Ford [26] to execute such queries. In order to return the actual cost of the shortest path, we allow `COST(p)` to be used in the `COLUMNS` clause, where `p` has been bound to a path.

```
SELECT cheapest.path, cheapest.cost
FROM GRAPH_TABLE (pg,
  MATCH CHEAPEST PATH p =
    (a:Person WHERE a.firstName = 'Alice')
      -[k:know COST 1/k.interactionCount]->*
      (b:Person WHERE b.firstName = 'Bob')
  COLUMNS (ELEMENT_ID(p) path, COST(p) cost)) cheapest;
```

Listing 4: Q2: cheapest path SQL/PGQ support in DuckPGQ

This third example query looks for the cheapest path between Alice and Bob, where the edge cost is proportionally lower between people that interact often with each other. Similar to `COST(p)` we use `ELEMENT_ID(p)` to turn a path into a list of `ELEMENT_ID`s (which as mentioned are DuckDB `ROWID`s in DuckPGQ).

3 DEMONSTRATION

This demonstration will use a laptop showing a python notebook interface with DuckDB on a large monitor, and a combination of prepared queries to interactively demonstrate the salient features of SQL/PGQ in general and DuckPGQ specifically (see also the submitted video). There will also be room for interactive query formulation by attendees.

The goal of the demonstration is to familiarize attendees with the new SQL/PGQ syntax, allow users to formulate queries with this new syntax, and show how these queries are executed in DuckPGQ. For the latter purpose, we will use supportive material in our poster, but also the output of `EXPLAIN PLAN` features of DuckDB to show how SQL/PGQ is mapped on the underlying primitives, and how the various operations perform.

The LDBC Social Network Benchmark dataset at various scale factors will be available to the user during the demonstration with SQL/PGQ queries from the BI [24] and Interactive [9] workloads.

Stretch goals of the demonstration are (i) to have the DuckPGQ extension module pre-compiled online for various platforms, so users will also be able to install it on their own devices to continue working with their own data (ii) to demonstrate some of our ongoing "zero-copy" graph access between DuckPGQ's CSR data structures and machine learning libraries; using SQL/PGQ to prepare and analyze GNN models.

ACKNOWLEDGMENTS

G. Szárnyas and P. Boncz were partially supported by the SQIREL-GRAPHS NWO project.

REFERENCES

[1] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases* 5, 3 (2013), 197–280.
[2] Renzo Angles et al. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* (2017).
[3] Renzo Angles et al. 2018. G-CORE: A Core for Future Graph Query Languages. In *SIGMOD 2018.*
[4] Renzo Angles et al. 2021. PG-Keys: Keys for Property Graphs. In *SIGMOD 2021.*
[5] Maciej Besta et al. 2019. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *CoRR* (2019).
[6] Peter A. Boncz et al. 2020. FSST: Fast Random Access String Compression. *PVLDB* 13, 11 (2020), 2649–2661.
[7] Alin Deutsch et al. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD 2022.*
[8] Till Döhmen et al. 2017. Multi-Hypothesis CSV Parsing. In *SSDBM.*
[9] Orri Erling et al. 2015. LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD 2015.*
[10] Nadime Francis et al. 2023. A Researcher's Digest of GQL (Invited Talk). In *ICDT 2023.*
[11] Michael J. Freitag and Thomas Neumann. 2019. Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates. In *CIDR 2019.*
[12] Per Fuchs et al. 2020. EdgeFrame: Worst-Case Optimal Joins for Graph-Pattern Matching in Spark. In *GRADES-NDA at SIGMOD 2020.*
[13] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR 2017.*
[14] André Kohn et al. 2022. DuckDB-Wasm: Fast Analytical Processing for the Web. *PVLDB* 15, 12 (2022).
[15] Laurens Kuiper and Hannes Mühleisen. 2023. These Rows Are Made for Sorting and That's Just What We'll Do. In *ICDE 2023.*
[16] Viktor Leis et al. 2014. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD 2014.*
[17] Viktor Leis et al. 2015. Efficient Processing of Window Functions in Analytical SQL Queries. *PVLDB* 8, 10 (2015), 1058–1069.
[18] Panagiotis Liakos et al. 2022. Chimp: Efficient Lossless Floating Point Compression for Time Series Databases. *PVLDB* (2022).
[19] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *BTW 2015 (LNI)*, Vol. P-241. GI, 383–402.
[20] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *SIGMOD 2018.*
[21] Adam Paszke et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *CoRR* abs/1912.01703 (2019).
[22] Pedro Pedreira et al. 2022. Velox: Meta's Unified Execution Engine. *PVLDB* 15, 12 (2022), 3372–3384.
[23] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *SIGMOD 2019.*
[24] Gábor Szárnyas et al. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *PVLDB* (2022).
[25] Daniel ten Wolde et al. 2023. DuckPGQ: Efficient property graph queries in an analytical RDBMS. In *CIDR 2023.*
[26] Manuel Then et al. 2014. The More the Merrier: Efficient Multi-Source Graph Traversal. *PVLDB* (2014).
[27] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *ICDE 2006.*