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 growing niche in data management. Many native graph systems and query languages have been created, but the functionality and performance still leave much room for improvement. The upcoming SQL:2023 will introduce the Property Graph Queries (SQL/PGQ) sub-language, giving relational systems the opportunity to standardize graph queries, and provide mature graph query functionality.

We argue that (i) competent graph data systems must build on all technology that makes up a state-of-the-art relational system, (ii) the graph use case requires the addition to that of a many-source/destination path-finding algorithm and compact graph representation, and (iii) incites research in practical worst-case-optimal joins and factorized query processing techniques.

We outline our design of DuckPGQ that follows this recipe, by adding efficient SQL/PGQ support to the popular open-source "embeddable analytics" relational database system DuckDB, also originally developed at CWI. Our design aims at minimizing technical debt using an approach that relies on efficient vectorized UDFs. We benchmark DuckPGQ showing encouraging performance and scalability on large graph data sets, but also reinforcing the need for future research under (iii).

1 INTRODUCTION

Graph Database systems have emerged as a growing niche in data management, with many property graph systems [7] such as Neo4j, TigerGraph, Dgraph, Titan and AWS Neptune becoming available, all using different query languages (i.e., Cypher, GSQL, GraphQL, Gremlin, SPARQL [2]). Property Graphs are directed graphs consisting of vertex and edge elements; where elements may have labels and associated key/value properties. Property graph systems are quite young, and performance of analytical queries on large graphs has been observed to be significantly lower than relational database systems, on graph queries that can also be formulated as SQL [16].

In RDBMS designs, there have been significant performance improvements in the past decade, with analytical systems such as Snowflake and Databricks adopting principles like skippable columnar storage with lightweight compression [24] (also popular in open-source formats such as Parquet and ORC), efficient load-balanced multi-core parallelism using "morsel-driven" scheduling [15] and efficient query execution techniques [14]: either using vectorized query execution or Just-In-Time low-level compilation of queries into executable programs.

The upcoming SQL:2023 introduces the **SQL/PGQ** (Property Graph Queries) sub-language [8], which allows (1) to define graph views over relational tables and (2) to formulate graph pattern matching and path-finding operations using a SQL syntax. These features narrow the functionality gap between RDBMSs and native graph systems, and unify the feature space with a common graph query sub-language, as PGQ is also a subset of the upcoming ISO Graph Query Language **GQL** [8] that native graph systems intend to adopt. GQL will add graph updates, querying multiple graphs and queries that return a *graph result*, rather than a binding *table*.

SQL/PGQ by example. If we have relational tables `Student` and `College` and connecting tables `know` and `enrol`, we can define a property graph `pg` consisting of `Person` vertexes connected to each other by edges with label `know` and to `College` vertexes via `studiesAt` edges:[1]

```
CREATE PROPERTY GRAPH pg
VERTEX TABLES(
 Student PROPERTIES(id,name,birthDate)   LABEL Person,
 College PROPERTIES(id,college))
EDGE TABLES(
 know  SOURCE Person KEY(id) DESTINATION Person KEY(id)
         PROPERTIES(createDate,msgCount),
 enrol SOURCE Student KEY(id) DESTINATION College KEY(id)
         PROPERTIES(classYear)           LABEL studiesAt)
```

In the below `SELECT` query the `MATCH` will bind variable `a` to all vertexes that satisfy a label-test `:Person` and have property `name='Ana'`. The comma separating the two pattern expressions implies a conjunction[2] with matching variable bindings: it requires `a` to also have an edge labeled `studiesAt` towards a `College c`:

```
SELECT study.college, study.pid FROM GRAPH_TABLE (pg,
  MATCH (a:Person WHERE a.name='Ana'),
        (a)-[:studiesAt]->(c:College)
  COLUMNS (c.college, ELEMENT_ID(a) AS pid)) study
```

The `MATCH` clause produces a conceptual *binding table* with each row holding matched bindings and one column for each variable. These bindings denote elements (e.g., a vertex or edge); the `COLUMNS` clause retrieves scalar values from those. The example retrieves the property `c.college` and the implicit element identifier[3] of `a`, as the columns of a temporary `GRAPH_TABLE` named `study` in the `FROM` clause.

 13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.

[1] The table name is the default label. DuckPGQ allows an additional `LABEL` list of max length 64, and a BIGINT `LABEL FROM col` *specifier* column. Elements only have a label from the list if their corresponding bit is set. This allows e.g., to express class membership with inheritance in labels. DuckPGQ will not support having the same label in multiple tables, as element patterns must always bind to a single table.
[2] Inside path expressions, the `|` will `UNION` pattern bindings, and `|+|` stands for `UNION ALL`; though neither is supported initially in DuckPGQ.
[3] `ELEMENT_ID()` is implementation-dependent; in DuckPGQ it returns a `rowid`.

Graph queries in PGQ are more concise than in pure SQL, with clear element syntax: `()` is a vertex and `[]` an edge. Inside element patterns one finds, in this order (all optionally): a variable, a `:` label-test – which can be grouped with `()` and composed using `&` (and), `|` (or) and `!` (not) – and a filtering `WHERE` clause. The edge requirement is also visual (`<-[]-` left, `-[]->` right, `<-[]->` both, `-[]-` any).

SQL/PGQ can match multi-step paths using a `{,}` quantifier (below: between 2 and 4 edges), and can also bind *paths* to variables (`p`), where a path is a list of alternating vertexes and edges, always starting and ending in a vertex. The below shows the *simplified syntax* of `-[:know]->{2,4}`. This compactly denotes a multi-edge path as a sequence of labels, inside slashes, with any quantifiers inline.

```
MATCH p=(a:Person)-/know{2,4}/->(b:Person)
```

A pure SQL version of this query would require to `UNION` 3 subqueries, each joining resp. 2, 3 and 4 aliases of the edge table in their `FROM` clauses. A quantifier `{x,}` without upper is bound is known as Kleene*. PGQ has shorthands `*` for `{0,}`, `+` for `{1,}` and `?` for `{0,1}`. In pure SQL, Kleene* requires a `RECURSIVE` query, which quickly becomes hard to write, read, and inefficient to execute.

```
MATCH ANY SHORTEST PATH p=(a:Person WHERE a.name='Ana')
  -[ e:know COST age(e.createDate)/max(1,e.msgCount) ]->*
                          (b:Person WHERE b.name='Bo')
```

`ANY SHORTEST PATH` was added to the path expression above (it binds `p` to 1 path of lowest cost); as PGQ Kleene* must have a finite result.

The *cost* of a path is the sum of the cost of its elements – with vertex cost 0 and edge cost 1 (i.e., cost=length), but element specifications can get an optional `COST` expression, allowing to search for *weighted shortest paths*.[4] The example above looks for the fastest messaging path between Ana and Bo. PGQ also can find `ANY k` shortest paths, i.e., `k=1` is default. Note that `ANY` is non-deterministic, as there may be more than `k` paths with equal cost. A deterministic alternative is looking for `ALL SHORTEST PATHS`. Paths can also be grouped in equivalence classes regarding their cost by using the `GROUPS` keyword, and then `ALL` paths belonging to the `k`-shortest classes can be deterministically returned. `ALL` Kleene* can also be made finite by constraining to `TRAIL` (no edge repeats), `ACYCLIC` (no vertex repeats) or `SIMPLE` (similar, but allows start=end). Finally, paths can be segmented using `[]` or `()`, allowing to bind variables to *sub-paths*, impose extra constraints, and express Regular Path Queries [23] by using quantifiers. Let's find all paths from Ana to Bo without vertex repeats (except maybe the last), over adults, that have the 5 shortest lengths, and bind `s` to those with the last hop cut off:

```
MATCH ALL 5 SHORTEST TRAIL GROUPS
 [ SIMPLE s=(a:Person WHERE a.name='Ana')
  [-/know/->(p:Person WHERE p.birthDate<'2004-01-01')]*]
   -/know/->(b:Person WHERE b.name='Bo')
```

Outline. In Section 2 we will outline our vision on how to create competent graph database systems, and in Section 3 bring this to bear in the blueprint of DuckPGQ: an extension module that supports most of SQL/PGQ in the open-source "embeddable analytics" database system DuckDB [21], originally developed at CWI. Specifically, this system allows for on-the-fly creation of CSR (Compressed Sparse Row) in-memory graph representations, made highly efficient by introducing a number of generic relational optimizations. We describe a minimal set of vectorized scalar user-defined functions (UDFs) that form the backbone of the graph-specific functionality for CSR creation and path-finding. The reliance on UDFs minimizes the impact on the mainline DuckDB code base, making DuckPGQ maintainable going forward.

In Section 4 we evaluate the performance of DuckPGQ, comparing it with property graph and purely relational systems, before outlining future research and conclusions in Section 5.

[4]This is a "language opportunity" in SQL:2023, but DuckPGQ supports it, allowing `COST(p)` in the `COLUMNS` clause to return path cost, as well as `ELEMENT_ID(p)` to return paths as a SQL list of `BIGINT` (rowids). DuckPGQ will not support `ALL` Kleene* path-finding, and initially only `ANY SHORTEST` single-edge Kleene* and no path grouping.

2 COMPETENT GRAPH SYSTEM DESIGN

We shortly outline 8 core features of competent analytical data systems design (**c1–8**) and then add 4 graph-specific features (**g1–4**).

c1: fast scans on elements with schema. Graph systems typically do not require upfront schema design. This is convenient for users, e.g., for quick prototyping and for evolving data mashups, but regrettably graph systems internally also tend to stay unaware of label and property structure of the elements they store. Systems that are unaware of structure, such as RDF systems, will turn a fast multi-column/property scan into many expensive joins between selections on a big table that stores all elements mixed together. Significantly increasing the amount of joins, in turn, exponentially increases the query optimization search space, leading to a more scant exploration when optimizing large queries and therefore worse query plans. Further, these avoidable joins often harbor (hard to detect) correlations that will throw off join cardinality estimates, further deteriorating plan quality. We argue that systems should detect the regularities of the data they store automatically [19] and exploit these for storage and query processing [20]. Note that SQL/PGQ systems are schema-aware by definition.

c2: skippable compressed columnar storage. Fast columnar scans, and table clustering and partitioning that allows data skipping based on pushed-down scan predicates using cheap min/-max statistics are a cornerstone of raw analytical performance [1]. Further, as columnar data has lower entropy than row data, compression tends to work well, reducing data volume, and hence I/O, network bandwidth, and RAM use – typically by a factor 3-4 [24].

c3: vectorized or data-centric execution. Analytical performance has been shown to improve by a factor 10–100× using either vectorized query processing or JIT code generation over traditional tuple-at-a-time interpreted execution [14]. Graph query processing subsumes relational functionality: scans, filters, grouping/aggregation and value-based joins are also functionalities of graph query languages, and such operations must be executed efficiently.

c4: morsel-driven multi-core. Modern hardware will often have tens of cores and a single heavy analytical query should benefit from near-linear scaling on these. This means that graph query languages should parallelize well and the state-of-the art here is flexible morsel-driven scheduling where a fixed number of threads pinned to the cores steal morsels of work (typically 10-100K data items) from a

queue, and exploit the scheduling flexibility provided by shared data structures (e.g., hash tables), for good load balancing [15].

c5: state-of-the-art query optimization. To get efficient query plans, one needs dynamic-programming-based query optimization, informed by good statistics: typically a combination of table samples (that allow to detect correlated predicates within a table) and *hyperloglogs* on most data to estimate distinct counts [17].

c6: bulk APIs/algebras. The interface between query operators should not be a single-value-at-a-time; but rather be framed in terms of *sets*. The popular TinkerPop is a key-value API: a graph navigation pattern containing 4 edges easily can lead to a million navigations, and thus API calls. In contrast, relational algebra is a good example of a bulk API. Bulk APIs amortize call overhead, provide opportunity for parallel IO and for parallel memory access (e.g., in vectorized hash-table or CSR lookups).

c7: out-of-core buffer manager. NVMe flash memory provides high bandwidth and low latency, allowing systems to reach almost-RAM performance on out-of-core data sizes. The Umbra approach (originating in LeanStore) [17] with low-overhead swizzling of disk references into memory pointers achieves this aim.

c8: explicit control over memory locality. Implementing graph storage using separate memory objects pointing to each other not only stands in the way of compressed columnar storage and leads to a bloated memory footprint, but also causes the system to lose control over physical memory locality as allocated objects are unlikely to be adjacent in physical memory (and this leads to increased CPU cache misses). This is compounded by working in so-called "managed" memory runtimes, such as in Julia or the JVM, where memory fragmentation and garbage collection will cause additional performance havoc over time. As such, we think a scalable graph system should be programmed in a language with explicit memory control and specifically optimize for memory locality.

The above principles are followed by modern analytical database systems such as Snowflake, Databricks Photon, ClickHouse and the academically developed Umbra, as well as our DuckDB. We argue that systems that follow these principles form a solid base for competent graph database systems, provided they add certain functionalities, important for graph workloads:

g1: fast CSR creation. The creation of a compact graph representation such as a CSR data structure is in itself a frequently desired functionality, e.g., exporting tabular data as a base for training Graph Neural Networks (GNNs). In addition, such a compact representation can be key to supporting efficient path-finding and worst case-optimal joins (see also g2-3). The ability to create it quickly on-the-fly has the additional advantage that one does not need to maintain this very write-unfriendly structure under updates.

g2: bulk path-finding. SQL/PGQ in the general case must perform path-finding between *sets* of source vertexes and *sets* of destination vertexes; yielding a multi-source multi-destination problem Such *bulk* path-finding provides opportunities for synergy between the individual (src, dst) path-finding tasks that a competent graph system must leverage. Bulk path-finding also provides a way to parallelize graph search over multiple cores (more on that later).

g3: worst-case-optimal joins (WCOJs). Multi-join algorithms can have provably better worst-case complexity than standard binary joins (between two tables) [18], e.g., WCOJs are $O(n^{1.5})$ on triangle queries, whereas plans with binary joins are $O(n^2)$. The benefits of WCOJs are strongest in queries where binary joins generate spurious intermediate results (i.e., much larger than the final result): when the joins are n:m and when the join graph has a cycle, since closing the cycle typically eliminates intermediates. This tends to occur in graph pattern matching, as edge-joins are n:m (explosive) and graph patterns often have cycles. Both efficient WCOJ algorithms [11], that e.g., fit the vectorized query execution model, as well as optimizer integration still require more research.

g4: factorized query processing. Materialization of explosive n:m joins can sometimes be postponed, by factorizing out the redundancies in such an exploded result [5]. This postponement can turn into largely avoiding the explosion, e.g., if a filter or an aggregation follows the joins. Integrating factorized query processing in data systems is relatively under-explored, has not been achieved in practical systems and should be on the research agenda.

We will now discuss the design and performance of DuckPGQ, that starts from the state-of-the-art analytical DuckDB system [21] that embraced **c1–8**, and adds **g1–2**; leaving **g3–4** for future work.

3 DESIGN AND IMPLEMENTATION

Since DuckDB is a popular system that is fast evolving, it is difficult to keep a fork in sync. Therefore, we made a design effort to implement PGQ as an extension module. DuckDB extensions can provide scalar UDFs. Scalar UDFs are as fast as builtin functions can be, and get invoked during vectorized expression evaluation, and thereby automatically profit from morsel-driven parallelism. DuckDB also allows extension modules to register parser extensions that are triggered by unknown SQL. Our DuckPGQ extension parses textual SQL:2023 queries with PGQ clauses – which is not understood by stock DuckDB, and translates this into a pure SQL query plan which gets executed by DuckDB as a normal query (with some UDF calls).

All SQL/PGQ pattern matching functionality, with filters, label tests etc. are trivially translated into equi-joins, unions, and filters. We also *prefer* to represent PGQ as normal SQL join plans, because we believe that relational and graph systems should not be separated: opportunities for WCOJ [11] (**g3**) and factorization [9] (**g4**) apply equally to tabular queries as to SQL/PGQ. The proper way is therefore to include those algorithms and optimization rules in the main relational engine [12].

The exception is Kleene*: we could translate it to `RECURSIVE` SQL, but it would be hard to express efficient shortest path-finding algorithms in that way. We chose to use Multi-Source (MS) BFS and Bellman-Ford algorithms [22] to support `ANY SHORTEST` path-finding, because in the general case, the start and end variables of a PGQ Kleene* get bound to sets (in its extreme, to *all* vertexes; so it would become an all-pairs problem), and these algorithms [22] get synergy out of resolving many shortest-path-finding problems. In order to efficiently execute these algorithms that need potentially very many navigational iterations (joins) we efficiently build a CSR on-the-fly (Listing 1), a functionality that is also useful for GNN data export.

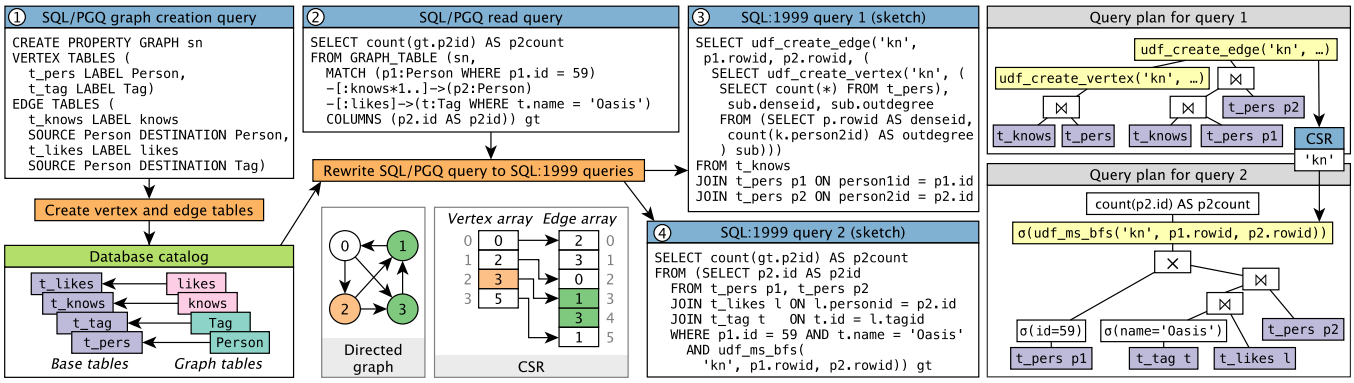


Figure 1: Example of a CSR data structure representing a directed graph and the workflow of evaluating a SQL/PGQ query using *shortest path-finding* in DuckPGQ. The SQL:1999 queries use the `udf_create_*` and `udf_ms_bfs` functions.

```
// done at udf_create_vertex CSR lookup (by the first
     thread to touch the CSR)
for (auto i = 0; i < csr_v_size+2; i++)
  csr_v[i] = 0; // init csr_v[] to zero

// vectorized create_vertex(int vid, int outdegree)
void udf_create_vertex(int64 csr_v[], int vectorsize,
     int64 vid[], int64 outdegree[]) {
  for (auto i = 0; i < vectorsize; i++)
    csr_v[vid[i]+2] = outdegree; // +2 is double sentinel
}

// done at udf_create_edge CSR lookup (by the first
     thread to touch the CSR)
for (auto i = 0; i < csr_v_size+2; i++)
  csr_v[i+1] += csr_v[i]; // running sum

// vectorized create_edge(int src, int dst)
void udf_create_edge(atomic<int64> csr_v[], int64
     csr_e[], int vectorsize, int64 src[], int64 dst[]) {
  for (auto i = 0; i < vectorsize; i++) {
    int pos = ++csr_v[src[i]+1]; // move write offset
    csr_e[pos-1] = dst[i]; // store dst at write offset
  }
}
```

Listing 1: Scalar UDFs that power parallel CSR creation

```
void udf_reachability(int vectorsize, int64 src[], int64
     dst[], bool result[])  {
  for (auto i = 0; i < vectorsize; i += 512)
    do512(min(512,vectorsize-i), src+i, dst+i, result+i);
}
void do512(int n, int64 src[], int64 dst[], bool res[]) {
  int512 visit[csr_v_size] = {0}, seen[csr_v_size] = {0};
  int512 next[csr_v_size], *x = next, *v = visit, *tmp;
  for (auto i = 0; i < n; i++) // visit sources
    visit[src[i]] |= (1 << i);
  while (ms_bfs(seen, v, x)) { tmp = x; x = v; v = tmp; }
  for (auto i = 0; i < n; i++) // reached dst?
    res[i] = (seen[dst[i]] >> i) & 1;
}
bool ms_bfs(int512 *seen, int512 *visit, int512 *next) {
  int512 active = 0;
  for (auto v = 0; v < csr_v_size; v++) // init
    { seen[v] |= visit[v]; next[v] = 0; }
  for (auto v = 0; v < csr_v_size; v++)
    if (visit[v]) // follow the edges of all active nodes
      for (auto e = csr_v[v]; e < csr_v[v+1]; e++) {
        int512 unseen = visit[v] & ~seen[csr_e[e]];
        if (unseen) next[csr_e[e]] |= unseen;
        active |= unseen;
      }
  return active != 0;
}
```

Listing 2: Scalar UDF implementing the MS-BFS algorithm

The execution starts (step ② in Figure 1) by generating sub-query (step ③) that counts the out-degree of all vertexes that are in play and obtains dense vertex numbers. Here we leverage DuckDB's rowids, which are almost-dense numbers, where the only possible holes are caused by tuple deletions. They are stable while the query runs and across queries as long as the table is not checkpointed. The renumbered vertexes are streamed into the `udf_create_vertex()`, which puts the degree in the `csr_v[]` array at the vertex number. These degrees get converted into offsets into `csr_e[]` by a running sum, with an extra leading zero, when `udf_create_edge()` initializes. The sub-query then also scans the relevant edges, converts logical keys to dense vertex numbers using two joins (that collect vertex rowids for src and dst) and streams that data into `udf_create_edge()`. This function adds the edge destinations at the proper place in the `csr_e[]` array. Because all of these UDF invocations happen in a parallelized query plan, it makes use of an atomic to increase the offset (the write position).

On the created CSR, a top-level sub-query in step ④ runs a UDF doing MS-BFS (Listing 2) or MS-BellmanFord. The gist is to execute a batch of path searches at the same time, using SIMD instructions. As DuckDB is vectorized, a call to e.g., `udf_reachability()` which uses MS-BFS, provides a vector (i.e., 1024) of such search pairs. Its state arrays `seen`, `visit` and `next` – denoting resp. already seen nodes, the current and next BFS frontier – hold one large integer for each vertex. Each large integer is a bitset, keeping one bit of state per search. The basic operations needed are OR, AND, NOT and zero-test; which AVX-512 can do for 512 bits in one CPU instruction. The algorithms thus create synergy between 512 searches computationally, but also because they share memory access (as sequential access to the CSR, and random access to the state arrays are done to benefit up to active 512 searches).

We also identified a number of generic SQL query optimizations (beyond **g3**–**4**, which are future work) that our use case can exploit. First: good-quality join order optimization is very important for graph pattern matching. DuckDB's 0.5.0 release introduces a major upgrade of its statistics, introducing *hyperloglog* statistics and better estimate propagation. This makes a large difference in DuckDB performance on TPC-H, TPC-DS, and LSQB [16]. Second, we note that we need to perform multiple identical joins, as the vertex table is

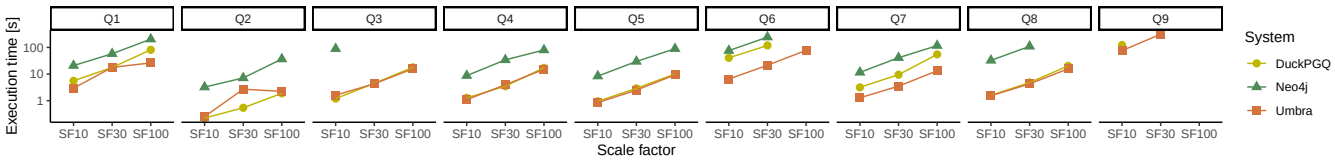


Figure 2: Pattern matching performance on the LSQB

joined three times. Therefore, we developed a generic optimization in DuckDB that shares a built hash-table in these cases. Third, we profit from the "perfect join": an optimization that changes a hash-join at run-time into an array-based lookup if during hash-build the keys turn out to be from an almost-dense numeric domain (rowids trigger this). We also profit from a similar "perfect" aggregation optimization in DuckDB, when computing the vertex degrees.

SF	Path-finding		LSQB	
	|Pers.|	|knows|	|V|	|E|
10	70k	2M	35M	217M
30	175k	6M	103M	650M
100	487k	23M	326M	2B
300	1M	68M	n/a	n/a
1 000	3M	227M	n/a	n/a
3 000	9M	670M	n/a	n/a

Table 1: Statistics of the graphs

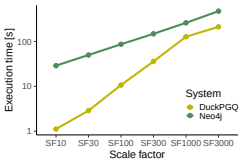


Figure 3: Path-finding performance

4 EVALUATION

We conduct an evaluation of path-finding and pattern matching performance on DuckPGQ, Umbra [11], and Neo4j, using data sets based on LDBC Social Network Benchmark social graphs [4].

Benchmark setup. We ran our experiments on a cloud instance running Fedora 36, equipped with 48 Intel Xeon Platinum 8375C vCPU cores, 248 GB RAM, and 2 NVMe SSDs in RAID-0 configuration. DuckPGQ ran embedded in a Python 3.10 process, while Neo4j and Umbra ran in Docker containers. For the path-finding experiments, we used DuckPGQ v0.2.2-dev7058, while for the pattern matching experiments, we used DuckDB v0.5.0-dev2374. For both experiments, we used Umbra version `bad073541` and Neo4j v4.4.2 Enterprise Edition (which has an experimental parallel runtime). When performing the benchmarks, we first load the database, and then execute the queries (on a cold database). The queries are run sequentially with a timeout of 1 hour for each query.

Path-finding performance. We used the Person–knows–Person subgraph of the LDBC SNB (Table 1). We defined a query that searches for shortest paths between a given set of source–destination pairs (Person ids) and finds paths in the graph between these Persons, returning the shortest path-length for each `(src, dst)` pair. We selected the input parameters such that they always result in a path because the Umbra `RECURSIVE` formulation of Kleene* crashes with an OOM for any case where a (short) path cannot be found. This uses a multi-source/destination bidirectional SQL formulation based on the SNB Interactive Umbra reference implementation[5].

For the system comparison experiment, we ran the query with 16K `(src,dst)` combinations, selected uniformly from 4M candidates. Figure 3 shows the results for scale factors 10 to 3 000. Umbra's `RECURSIVE` implementation crashes with OOM. We were able to get (slow) results with 2K `(src,dst)` pairs, only on the smallest SF10. Neo4j completes the workload on all scale factors, with good performance, thanks to its bi-directional path-finding algorithm. For the moment, the SIMD-friendly MS-BFS in DuckPGQ is uni-directional, but is able to generally beat Neo4j still.

Pattern matching performance. We used the *Labeled Subgraph Query Benchmark* (LSQB) [16] to assess the performance of pattern matching. The LSQB data set (Table 1) contains labeled graphs based on the LDBC SNB social network graph [4]. LSQB defines 9 queries, each counting the occurrences of a given graph pattern using labels such as Person, Tag, knows, and likes. Queries 1–6 are *basic graph patterns* [2] which can be expressed by equi-joins: (Q1) long path, (Q2) simple cycle, (Q3) triangle, (Q4) star, (Q5) low-cardinality path, (Q6) high-cardinality path. Queries 7–9 extend queries into *complex graph patterns* [2] by adding optional and negative edges, corresponding to outer- and anti-joins, resp.: (Q7) Q4 with optional edges, (Q8) Q5 and (Q9) Q6 with a negative condition.

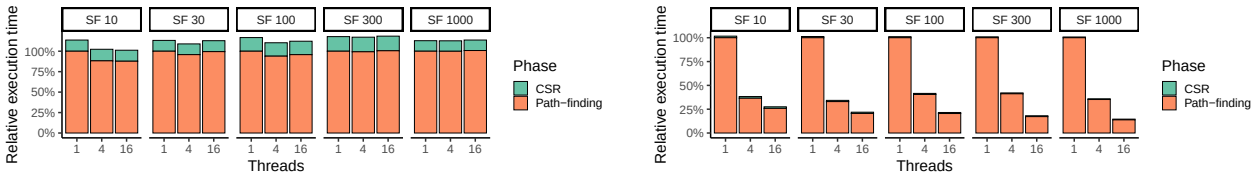
Figure 2 has the results for scale factors 10 to 100. The execution times of Umbra and DuckPGQ are within an order of magnitude for all queries. The two RDBMSs constantly outperform Neo4j which is unable to finish 4 queries on SF100.

The results show that Q6 and Q9 are the most difficult queries: only Umbra was able to complete Q6 on SF100 and Q9 on SF30 – in this query it uses its WCOJ [11]. No system could run Q9 on SF100. These queries define long explosive paths, terminating in an aggregation, that will benefit from factorization techniques [6] (which are currently not implemented in any practical DBMS). These results demonstrate the need to research **g3-4**.

Bulk path-finding performance in DuckPGQ. The bulk path-finding work performed by MS-BFS and variants for (weighted) reachability, path length and path retrieval in DuckDB, make use of UDFs. We are interested in the questions (i) is on-the-fly CSR creation a bottleneck? and (ii) how well does parallelism work (both for (a) CSR creation and (b) path-finding)?

Regarding CSR creation and question (iia), parallel scaling works well. As a result, (i) CSR creation is never a bottleneck if there is significant path-finding work. However, if there is little path-finding work, in an extreme case just a single `(src,dst)` pair of vertexes, then its overhead can be significant. In Figure 4 we show results comparing CSR creation overhead comparing 1 morsel (the minimum) of path-finding work with 16 morsels (the amount of real cores we have). While the overhead is more significant with 1 morsel, it still does not dominate then. Regarding question (iib) how well parallelism works for path-finding, there is good and bad news. In heavy duty scenarios we see parallel scaling, albeit not fully linear: a factor 6 on 16 threads. This non-linearity is probably related to multi-core path-finding becoming memory-bound. However, in

[5]https://bit.ly/github-snb-umbra-kleene



(a) 1K source-destination pairs with 1 morsel

(b) 16K source–destination pairs with 16 morsels

Figure 4: Relative performance of the MS-BFS UDF between using 1 morsel and 16 morsels, executed on 1, 4, and 16 threads

situations where there is just 1 morsel, the expected outcome is confirmed: there is work for just one core and hence no scalability.

Takeaways for SQL/PGQ. The proposed SQL/PGQ will significantly increase the usability of SQL systems on graph use cases. It has truly been integrated into the SQL syntax, and has been designed by ISO in liasion with LDBC, and is partially founded on its G-CORE [3] design, but also on formal studies of other graph query languages [2, 10]. Still, while the SQL/PGQ proposal ensures *finite* query results, in our vision a focus on *interactive* query results would be most useful. Specifically, `ALL ACYCLIC PATHS` on a Kleene* is finite because there are finite vertexes in graph databases; but in a large connected component there will be typically exponential amounts of such paths w.r.t. its size, making termination of such queries on large graphs unlikely, certainly in interactive time. Features that typically lead to a hanging server or a large cloud bill will not be appreciated by most users. In the future, DuckPGQ may support `ALL` path-finding on `TRAIL`, `SIMPLE` and `ACYCLIC` paths but only on strongly bounded quantifiers, which can be translated into plain unions, joins and filters. Another problem apparent in SQL/PGQ is that `SHORTEST TRAIL (a:A)-/e*/-(b:B)-/e*/(c:C)` breaks the quite useful decomposability into `SHORTEST PATH (a:A)-/e*/-(b:B)` and `SHORTEST PATH (b:B)-/e*/(c:C)` present in default `WALK` semantics. In other words, we find path constraints that span multiple Kleene* problematic to support, for only a small functional value-add.

5 CONCLUSION AND FUTURE WORK

We outlined the semantics of the SQL/PGQ, the main novelty in SQL:2023, and made the case for competently addressing graph database architecture by building on techniques from analytical relational technology. Putting these two together, we presented the design of DuckPGQ, an inobtrusive extension of DuckDB, CWI's embeddable analytics system. SQL/PGQ can be largely mapped onto relational queries, and we identify a number of relational optimizations that can be useful to such queries. Kleene* path-finding, be it for reachability or for (weighted) path length or path retrieval can theoretically be formulated as `RECURSIVE` queries, as we do in Umbra; but our experiments show that this is slow and brittle. DuckPGQ introduces bulk path-finding by adopting SIMD-friendly multi-source algorithms, as well as an on-the-fly compact in-memory graph representation (CSR), with an implementation in scalar UDFs. This system is able to beat the Enterprise version of Neo4j on both pattern matching and path-finding; and is comparable to Umbra on the former, thanks to recent optimizer improvements.

As for future work, we recommend **g3**+**g4**: better integration of (vectorized) WCOJ and integration of practical factorized query processing in analytical relational systems. Regarding path-finding algorithms, we did notice limitations of our parallelism model: it only works well if there are enough `(src,dst)` work tuples, because morsel-driven parallelism is tuple-driven and without tuples there is not enough parallelism. While it is hard to effectively parallelize Dijkstra, it is known to be possible to effectively parallelize our multi-source algorithms, running individual searches in parallel, by partitioning work on the vertexes [13]. However, it is a research challenge for database architectures to reconcile this elegantly with any query-pipeline-driven parallelization method.

REFERENCES

[1] Daniel Abadi et al. 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*.
[4] Renzo Angles et al. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020).
[5] Nurzhan Bakibayev et al. 2012. FDB: A Query Engine for Factorised Relational Databases. *Proc. VLDB Endow.* (2012).
[6] Nurzhan Bakibayev et al. 2013. Aggregation and Ordering in Factorised Databases. *Proc. VLDB Endow.* 6, 14 (2013), 1990–2001.
[7] Maciej Besta et al. 2019. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *CoRR* abs/1910.09017 (2019).
[8] Alin Deutsch et al. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD*.
[9] Daniel Flachs, Magnus Müller, and Guido Moerkotte. 2022. The 3D Hash Join: Building On Non-Unique Join Attributes. In *CIDR*.
[10] Nadime Francis et al. 2018. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD*.
[11] Michael J. Freitag et al. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.* 13, 11 (2020), 1891–1904.
[12] Pranjal Gupta et al. 2021. Columnar Storage and List-based Processing for Graph Database Management Systems. *Proc. VLDB Endow.* 14, 11 (2021), 2491–2504.
[13] Moritz Kaufmann et al. 2017. Parallel Array-Based Single- and Multi-Source Breadth First Searches on Large Dense Graphs. In *EDBT*.
[14] Timo Kersten et al. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (2018), 2209–2222.
[15] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*.
[16] Amine Mhedhbi et al. 2021. LSQB: A large-scale subgraph query benchmark. In *GRADES-NDA at SIGMOD*.
[17] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.
[18] Hung Q. Ngo et al. 2013. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.* (2013).
[19] Minh-Duc Pham et al. 2015. Deriving an Emergent Relational Schema from RDF Data. In *WWW*.
[20] Minh-Duc Pham and Peter A. Boncz. 2016. Exploiting Emergent Schemas to Make RDF Systems More Efficient. In *ISWC*.
[21] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *SIGMOD*.
[22] Manuel Then et al. 2017. Efficient Batched Distance, Closeness and Betweenness Centrality Computation in Unweighted and Weighted Graphs. *Datenbank-Spektrum* 17, 2 (2017), 169–182.
[23] Sarisht Wadhwa et al. 2019. Efficiently Answering Regular Simple Path Queries on Large Labeled Networks. *SIGMOD*.
[24] Marcin Zukowski et al. 2006. Super-Scalar RAM-CPU Cache Compression. In *ICDE*.