Master Thesis

Integrating SQL/PGQ in DuckDB

by

Daniël ten Wolde
(2619049)

First supervisor:	Peter Boncz
Daily supervisor:	Gábor Szárnyas
Second reader:	Gábor Szárnyas

May 31, 2022

Submitted in partial fulfillment of the requirements for
the joint UvA-VU degree of Master of Science in Computer Science

Integrating SQL/PGQ in DuckDB

Daniël ten Wolde
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
d.l.j.ten.wolde@student.vu.nl

ABSTRACT

This thesis will focus on implementing path finding algorithms in the open-source in-process SQL OLAP database management system DuckDB. This is done to further complete the integration of SQL/PGQ, an extension of SQL, in DuckDB. The algorithms, one of which is a batched version of Bellman-Ford, will make it possible to retrieve the path length between any two nodes in a (weighted) graph. Batched Bellman-Ford and other algorithms, will be implemented as user-defined functions in DuckDB, which make for a lightweight approach that does not alter the internal workings of DuckDB. In addition, possible optimizations in DuckDB for graph-like queries will be identified and implemented.

1 INTRODUCTION

There is a growing desire to perform more complex analyses on the increasing amounts of data being gathered. This data is connected, which will often be represented and thought of as a graph [47]. An extended survey of Sahu et al. [45] shows that graphs are used across a very diverse range of domains. Graphs often provide a natural way to structure data involving entities, represented as vertices, and the relationships between them, represented as edges. In turn, this increased desire has caused an increased rise in the attention given to graph database management systems (GDBMSs) [20]. These systems provide a new way of visualising and analysing graph data. However, the performance of these systems often leaves to be desired [20, 40]. The question is whether there is a need for this new method of looking at the data. Traditional relational database management systems (RDBMSs) are perfectly capable of storing graph data by using a vertex table and an edge table. Still, providing a way to perform these more complex analyses has been difficult up to now due to the limitations of the standard query language SQL [37].

In response to the limited capabilities, a plethora of GDBMSs arose, each having its own graph query language [9]. Examples of systems and their query languages are TigerGraph with GSQL [56], Neo4j with Cypher [21], Oracle Labs PGX with Property Graph Query Language (PGQL) [57]. Amazon Neptune even supports three distinct languages, namely OpenCypher, Gremlin, and SPARQL. Within these graph query languages, it is often easier to write queries containing graph pattern matching and path finding. However, each query language has a different syntax, different semantics, and different capabilities. These problems make working with graph-based data cumbersome for users.

The limited capabilities of SQL will change, as an extension of SQL is scheduled to release in March of 2023 called SQL/Property Graph Query (SQL/PGQ), making it easier to support this new type of workload related to graph data in RDBMSs [34].

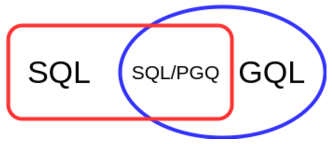

Figure 1: SQL/PGQ in relation to SQL and GQL.

For querying graph data, two functionalities are deemed most important: *graph pattern matching* and *path finding* [6]. In SQL it is possible to write queries containing graph pattern matching and path finding, but, these queries are often hard to write, understand, and inefficient to evaluate [28]. In particular, path finding requires using recursive queries. An example can be found in Listing 8, which shows a path finding query that returns all paths starting from *Person 1* [19]. Support for this was added in SQL:1999 [35], and is supported by popular RDBMSs such as PostgreSQL and SQLite. However, as shown by Michels and Witkowski [37], even relatively simple graph queries in plain SQL take up many lines and are more difficult to understand compared to the equivalent query in SQL/PGQ. The goal of SQL/PGQ is to provide a more compact syntax for graph-like data and make path finding queries, such as reachability or shortest path, more accessible.

Work on a standardization of a graph query language started in 2018 [5, 18, 24]. Currently the *ISO/IEC JTC1 SC32 WG3 Database Languages* working group is developing SQL/PGQ, which will become part of the upcoming SQL:2023 standard [34]. In SQL/PGQ a graph can be defined in terms of tables [36] and queries can contain special syntax for path finding and graph pattern matching. The standard is limited to read-only queries, as it will not be possible to modify the graph through the graph tables in SQL/PGQ. Therefore, the same workgroup is also working on Graph Query Language (GQL) [33], in which it will be possible to modify the data in addition to all the features contained in SQL/PGQ. GQL will be a superset in relation to SQL/PGQ, see Figure 1, which will have the ability to manipulate the data.

In this thesis we integrate the SQL/PGQ standard in DuckDB. DuckDB is an open-source in-process SQL OLAP database management system produced [42] by CWI [30]. With the new SQL/PGQ standard releasing soon, work on integrating it into DuckDB has already started by Singh et al. [48]. However, the standard has not been fully integrated as of now. In particular the shortest and cheapest path functions are not yet implemented. Additionally, the feature to return the nodes contained in these cheapest and shortest path functions will also be implemented. Therefore, the goal is to complete this integration by adding these functionalities. Additionally, we will identify and implement various optimizations especially related to typical graph queries in DuckDB.

The structure of this thesis proposal is as follows. Section 2 discusses relevant background information regarding graphs and SQL/PGQ. In Section 3 we formulate the research questions and the goals of this thesis. Section 4 describes how the research questions will be answered and evaluated and provides a research plan. We end with a conclusion in Section 7.

2 BACKGROUND

2.1 Property Graph Data Model

Graphs can be used to model complex, connected data through the use of vertices and edges. The vertices, sometimes referred to as nodes, represent objects. The edges, also referred to as relationships, represent the relations between objects. The simplest form of a graph is the *simple directed graph model* [9]. In this model, the graph consists of a set of vertices, and a set of edges. Each edge has a source vertex and a destination vertex. In the case of an undirected graph, both vertices within an edge act as the source and destination vertex. Graphs can either be weighted or unweighted. In weighted graphs, an edge between two vertices contains a weight. This weight can be different for every edge. In unweighted graphs, all edges are of equal weight. This is important when determining the shortest or cheapest path between two vertices in a graph. In the case of an unweighted graph, the shortest path will also be the cheapest path, since all weights are equal. For a weighted graph, this need not be the case.

The simple graph model suffices in certain cases, such as computing the reachability of a vertex from all other vertices. However, it is not possible to store any information in either the vertices or edges. For this we introduce the *Labelled Property Graph (LPG)* model, which is often used in graph databases [9]. The simple graph model is now enriched with the ability to assign labels to both vertices and edges. A label can for example be 'Person', or 'Knows'. Depending on the specific implementation of the graph database there can either be one or multiple labels assigned to a vertex or edge [5]. In addition to labels, there can also be properties assigned to vertices and edges. They can contain more specific information to the vertex or edge than the label. For example, a property could be (Age, 23) or (Name, Daniël). Examples of database systems that have based their data model on the LPG model are Neo4j [43], TigerGraph [56], and Oracle PGX [32].

2.2 SQL/PGQ

SQL/PGQ limits itself to read-only graph queries and how to define graph views over a tabular schema [18]. The two most important graph querying functionalities are graph pattern matching and path finding as described by Angles et al. [6]. These functionalities become more accessible with the addition of SQL/PGQ and queries involving these can be more easily expressed [28, 37].

With SQL/PGQ, graphs are stored as a set of vertex tables and edge tables, where each row in a vertex/edge table represents a vertex/edge in the graph [12]. A graph can be defined using the SQL statement [49] found in Listing 1.

```
CREATE PROPERTY GRAPH <name> [WITH SCHEMA <schema>] [FROM
      <subquery>]
```

Listing 1: Creating a graph in SQL/PGQ

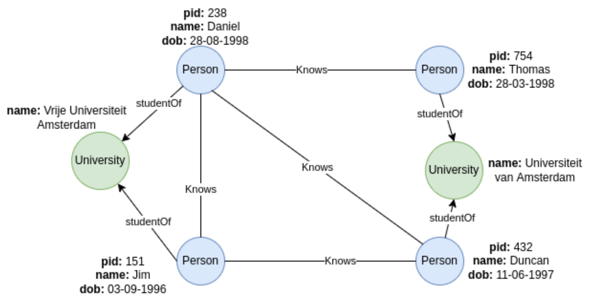

Figure 2: Graph of friend network and where they study.

For example, if we wish to create the graph of Figure 2, symbolizing a group of friends who all studied at some university, we would use the following query:

```
CREATE PROPERTY GRAPH friend_network
VERTEX TABLES ( Person PROPERTIES ( name, dob ),
                University PROPERTIES ( name ) )
EDGE TABLES (   knows SOURCE Person DESTINATION Person NO
      PROPERTIES,
                studentOf SOURCE Person DESTINATION
     University NO PROPERTIES )
```

Listing 2: Creating a friend network graph in SQL/PGQ

To match a pattern to this graph in SQL/PGQ, the `MATCH` syntax can be used [18], as can be seen in Listing 3. For example, the following selects the name and date of birth from persons from the graph in Figure 2 whose name is equal to 'Daniel':

```
SELECT p.name, p.dob
FROM Person GRAPH_TABLE (
    MATCH ( a:Person WHERE a.name = 'Daniel' )
    COLUMNS (
        a.name,
        a.dob )
    ) p
```

Listing 3: Pattern matching all nodes with the property name Daniel

Matching such a simple graph pattern is also relatively straightforward in plain SQL, however, it could be argued that the SQL/PGQ syntax feels more natural to write than the plain SQL one, since it is closer to how the human mind tends to interpret the world regarding objects and their connections [44]. As can be seen in Listing 3, we use the () notation to address a node. Addressing an edge can be done by using `[]`. To indicate that an edge is pointing from source to destination we use `(source)-[edge pattern]->(destination)`. This ASCII-art style is inspired by the syntax of Cypher, one of the most widely used graph query language [41].

A more complex graph pattern, involving both nodes and undirected edges could look like the following:

```
MATCH ( a:Person WHERE a.name = 'Daniel' )-[ e:knows ]-(
     b:Person )-[ f:studentOf ]-( c:University )
```

Listing 4: Pattern matching using nodes and edges

This statement would extract all patterns that match node *a* being a Person with the name Daniel, who knows a Person who is a studentOf a University. Within every node or edge, we can filter the possibilities by adding a WHERE statement, as can be seen in Listing 4.

One of the features of SQL/PGQ will be the ability to match a single edge pattern or a parenthesized path pattern for an arbitrary length [18]. An example where we want to find paths of length 2 to 5 of *knows* edges:

```
MATCH ( a:Person )-[ e:knows ]->{2,5}( b:Person )
```

Listing 5: Path length of 2 to 5 knows edges

Finding such a path in plain SQL is significantly more difficult [37]. SQL/PGQ is not limited to quantifying the upper-bound of the path length in such a path finding query. Similar to regular expressions, it is possible to use the Kleene star (*) operator, to indicate that the pattern can occur 0 or more times. Additionally, matching the pattern 1 or more times is possible using the Kleene plus (+) symbol. The following is an example of a pattern using the Kleene star operator:

```
MATCH ( a:Person )-[ e:knows ]->*( b:Person )
```

Listing 6: Path length of arbitrarily many knows edges

Another addition in SQL/PGQ will be the ability to return the path corresponding to a query. Alongside returning every path that adheres to a given pattern, it will be possible to return the cheapest or shortest path. The cheapest path will return the path in which the total weight of the edges along the path are the lowest. The shortest path returns the path containing the least amount of hops from the starting node to the end node. In case the graph is unweighted, cheapest and shortest path will provide equivalent results, since the weight of any edge will be equal to 1.

2.3 Graph traversal algorithms

2.3.1 Multi-Source Breadth-First Search (MS-BFS). In order to compute the shortest path of unweighted graphs, the batched variant of the MS-BFS algorithm developed by Then et al. [54] can be used. The pseudo-code of the algorithm is provided in Algorithm 1. The algorithm is an example of a bulk algorithm that fits well in the vectorized execution engine of DuckDB [11]. It is able to run multiple BFSs concurrently on the same graph in a single CPU core. Furthermore, it can make use of Single Instruction Multiple Data (SIMD) instructions, such as AVX-512 [26], that are available in modern CPUs. This lets us handle 512 BFS steps in one CPU cycle, further increasing the efficiency in CPU usage. Additionally, it has the ability to scale up as the number of CPU cores increases. Since there are no dependencies between the various BFSs, they can be divided over multiple cores. It makes use of the small-world property [2] that occurs when the diameter of a graph is small in comparison to the total number of nodes. This means that each BFS discovers most vertices in a few iterations, and concurrent BFSs have a high chance of overlapping sets of discovered edges in the same iteration. This allows access to be shared among the multiple BFSs and reduce the chance of cache misses, reducing the overall computation time [54].

2.3.2 Cheapest path. To find a cheapest path in weighted graphs either the Dijkstra or Bellman-Ford algorithms can be used. The Dijkstra algorithm has a worst-case time complexity of $O(|E| + |V|\log|V|)$ [22], which is better than Bellman-Ford's worst-case time complexity of $O(|V|\cdot|E|)$ [7]. However, the expected runtime of Bellman-Ford is $O(|E|)$ in large dense graphs with low diameter [60]. Then et al. [53] propose a Batched Bellman-Ford-based algorithm, shown in Algorithm 3, to find the shortest distance between two nodes in a graph. This algorithm can make use of SIMD instructions to increase the CPU usage efficiency, which is not possible with the standard Bellman-Ford algorithm [8], which is shown in Algorithm 2.

Algorithm 1 MS-BFS

```
    Input: $G, \mathbb{B}, s$
2:  $seen_{s_i} \leftarrow \{b_i\}$ for all $b_i \in \mathbb{B}$
    $visit \leftarrow \bigcup_{b_i \in \mathbb{B}} \{(s_i, \{b_i\})\}$
4:  $visitNext \leftarrow \varnothing$
    while $visit \neq \varnothing$ do
6:      for each $v$ in $visit$ do
            $\mathbb{B}'_v \leftarrow \varnothing$
8:          for each $(v', \mathbb{B}') \in visit$ where $v' = v$ do
                $\mathbb{B}'_v \leftarrow \mathbb{B}'_v \cup \mathbb{B}'$
10:         end for
            for each $n \in neighbours_v$ do
12:             $\mathbb{D} \leftarrow \mathbb{B}'_v \setminus seen_n$
                if $\mathbb{D} \neq \varnothing$ then
14:                 $visitNext \leftarrow visitNext \cup \{(n, \mathbb{D})\}$
                    $seen_n \leftarrow seen_n \cup \mathbb{D}$
16:                 Do BFS computation on $n$
                end if
18:         end for
        end for
20:     $visit \leftarrow visitNext$
        $visitNext \leftarrow \varnothing$
22: end while
```

Algorithm 2 Bellman-Ford

```
    Initialize-Single-Source(G,s)
2:  for $i \leftarrow 1$ to $|V[G]| - 1$ do
        for each edge $(u, v) \in E[G]$ do
4:          Relax(u, v, w)
        end for
6:  end for
    for each $edge(u, v) \in E[G]$ do
8:      if $d[v] > d[u] + w(u, v)$ then
            Return False
10:     end if
    end for
```

2.4 DuckDB

DuckDB is a database management system specialized in OLAP workloads [42]. This means that the system is optimized more towards analytical queries, touching large data volumes using joins and aggregations. Just like SQLite, DuckDB is an in-process system, though SQLite is specialized in OLTP workloads.

DuckDB consists of a number of components: Parser, logical planner, optimizer, physical planner, and execution engine. The

Algorithm 3 Directed batched Bellman-Ford-based algorithm

```
Input: WeightedGraph G, Array<Vertex> sources
Output: VertexProperty<BatchVar<double> > dists
VertexProperty<BatchVar<bool> > modified = false
dists = Infinite
for i=1..sources.length do
    Node v = sources[i]
    dists[v][i] = 0
    modified[v][i] = true
end for
bool changed = true
while changed do
    changed = false
    for each v in G.vertices do
        if not modified[v].empty() then
            for each v in G.neighbours(v) do
                double weight = edgeWeight(v,n)
                for each i in modified[v] do
                    double newDist = min(dists[n][i], dists[v][i]
+ weight)
                    if newDist != dists[n][i] then
                        dists[n][i] = newDist
                        modified[n][i] = true
                        changed = true
                    end if
                end for
            end for
        end if
    end for
end while
```

system can be accessed through a C/C++ API, as well as a SQLite compatibility layer. The SQL parser is based on the PostgreSQL SQL parser [42]. The logical planner consists of a binder and a plan generator. The binder is responsible for the expressions from the query related to anything containing a schema such as tables and views and retrieves the required columns and data types. The plan generator then creates a tree of basic logical query operators from the retrieved parse tree. Once the logical planner is done, the optimizer is used to optimize the logical plan. This will result in an optimized logical plan which is given to the physical planner where it is turned into a physical plan. The physical plan consists of operators, where each operator implements one step of the plan. An example of a unary operator is the *scan*, which scans a table and brings each tuple of a relation into main memory [23]. A join operator that makes use of two tables is an example of a binary operator.

These operators are split up into pipelines, which determines the order of operation execution. A query can consist of one or more pipelines, some of which contain a dependency on another pipeline. A pipeline with a dependency is for example, one containing a *join* operator. The start of a pipeline is referred to as a source and the end is referred to as the sink, which is where all the data is collected (materialized). Only sink operators, such as sorting, building a hash table, and hash aggregation need to see all the data before they can proceed. All other operators do not need to materialize all data before proceeding. In the case of binary operators there are always two pipelines, one that builds the hash table, and one that probes this hash table. Both pipelines contain a source and a sink, and since the probing is a non-materializing operation it can be scheduled in the middle of a pipeline.

With every join, a hash table needs to be built on which the join can be performed and the operator needs to wait for the entire hash table to be built. Only then can the join be correctly executed. In the same vein, another pipeline might require the outcome of this join before it can be executed, creating a chain of dependencies.

DuckDB makes use of hash tables to perform join operations[1]. Cardinality estimation is performed to asses which of the two tables is the smaller one. These estimations can be wrong, so it is not guaranteed that the smallest table is always used.

This smallest table is then used to build a hash table from, this is also referred to as the *sink* or the build side. The other, larger, table is then used to probe the hash table, looking for matching entries, this is referred to as the *source* or the probe side. Whenever the two tables are of equal size, a random one of the two is chosen to be the sink.

The execution engine of DuckDB is vectorized [42]. The use of vectors is more CPU efficient than the more common tuple-at-a-time execution found in other DBMSs [13]. ACID-compliance (**A**tomic, **C**onsistent, **I**solated, and **D**urable) in DuckDB is provided by Multi-Version Concurrency Control [42]. To allow for persistent storage, DuckDB uses the DataBlocks storage layout, which is read-optimized [42]. A useful feature of DuckDB is the allowance of extension modules, also referred to as Scalar user-defined functions (UDFs). These Scalar UDFs are as fast as the built-in functions of DuckDB due to the vectorized query processing, meaning that the parallelization of the UDF is handled by DuckDB.

2.5 Current state of SQL/PGQ in DuckDB

This thesis will make use of the work done by Singh et al. [48]. They identified several challenges that needed to be addressed. DuckDB is primarily intended for tabular workloads and wants to limit its core features to those required for the tabular types of workloads. Therefore, minimal changes to the parser and transformer were made to allow correct parsing of SQL/PGQ queries. One of the first challenges to be tackled was successfully parsing SQL/PGQ queries. Modifications were made to the DuckDB parser to allow for the ASCII-art style query syntax that is introduced with SQL/PGQ as shown in Listing 4. Additionally, new statements like GRAPH, LABEL, PROPERTIES were added to the parser to allow for correct parsing of SQL/PGQ queries. The SQL/PGQ queries are transformed into plain SQL queries using Scalar UDFs, an example of which can be seen in Listing 9 and Listing 10.

Some queries are more difficult to translate from SQL/PGQ to plain SQL. For instance, queries containing the Kleene star operator, shown in Listing 6 are challenging to translate as discussed by Michels and Witkowski [37]. The translation requires to make use of the recursive *WITH* statement. The translation for queries containing the Kleene star operator is not always correct at the time

[1]Whenever the ids of the smaller table are dense, meaning the maximum id is not much larger than the size of the table, an array is used instead, eliminating the need to build a hash table. This is referred to as a perfect join [1].

of writing. Therefore, one of the tasks for this thesis is to improve this translation and ensure its correctness.

Another consequence of limiting the core features was the decision to implement the new operators, such as the batched MS-BFS algorithm [54] described in Section 2.3, as Scalar UDFs. With MS-BFS it is possible to compute the reachability of any number of nodes given one or more source nodes. DuckDB handles the parallelization for Scalar UDFs using the morsel-driven method [29], which helps when scaling the batched MS-BFS algorithm. Another benefit of implementing these operators as Scalar UDFs instead of expressions is the fact that little to no changes have to be made to the internals of DuckDB. To implement reachability, no further changes needed to be made to the parser and no new logical and physical operators had to be introduced.

Another challenge Singh et al. looked at was the access patterns. Typically, a pointer-based data structure can be used for $O(1)$ lookup. In this case, every node contains a mini-index to all nearby nodes [58]. However, this becomes inefficient in the case of multiple traversals due to poor memory locality. An alternative is to make use of a Compressed Sparse Row (CSR) data structure. With this data structure, the rowids of both the vertex and edge tables are condensed into integers in the range $[0, |V|)$ and $[0, |E|)$. Most important are the edge tables, which are usually much larger, leading to a higher chance of poor memory locality. Thus, a Scalar UDF was introduced to create the CSR data structures. The CSR structure needs to be created for both the vertex and the edge table in order for the reachability function to work.

A weighted directed graph is represented in Figure 3. Vertices that share an edge contain a weight, ranging between $[0, 1)$ in this example. For example, the edge from vertex 1 to vertex 2 has a weight of 0.3. Figure 4 shows the CSR representation for the graph in Figure 3. The *Vertex array* contains offsets for the *Edge array*. The Edge array holds the indices of destination vertices. To find all outgoing edges from a given source vertex, we take its index. The offset at the index of the Vertex array gives us the offset for the first outgoing edge in the Edge array and subsequently the index of the destination vertex. For example, in Figure 3 we observe the first edge from vertex 1 directs to vertex 2. For vertex 4, the first edge direct to vertex 1, therefore, the offset of the Vertex array at index 4 points to 1 in the Edge array. The offset between two Vertex array offsets can be used to determine the number of edges for a source vertex. This condenses the matrix-like data structure into regular arrays, improving the memory locality. Additionally, the structure only needs to be created once, after which it is kept in memory. The work by Singh et al. only supports unweighted graphs. Thus, one of the goals is to extend the CSR structure by also supporting weighted graphs.

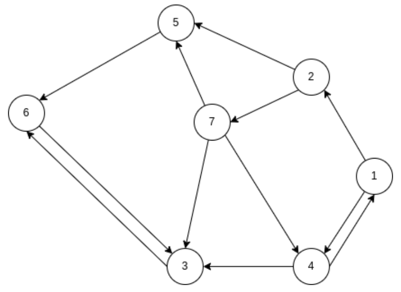

Figure 3: An example of an unweighted graph

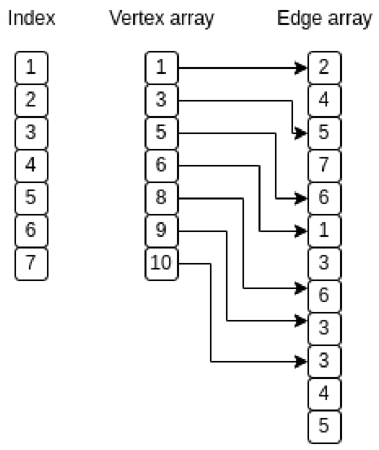

Figure 4: CSR representation of the graph in Figure 3

2.6 LDBC Social Network Benchmark

The Linked Data Benchmark Council (LDBC) has created the Social Network Benchmark to test various functionalities of graph-like database management systems [4]. The benchmark contains datasets structured similarly to real-world social networks. Two workloads are described in the benchmark, the Interactive workload which focuses on interactive transactional queries, and the Business Intelligence workload that focuses on analytical queries. The latter is more relevant to this thesis as DuckDB is specialized towards analytical (OLAP) workloads [42], thus making the workload better suited.

The Business Intelligence workload consists of complex read queries and refresh operations (insert and delete operations). Since SQL/PGQ is a read-only query language, we will only be focussing on the queries related to the read operations, of which there are 20. Each query contains a number of parameters which are generated before executing the query. These parameters are then used in the query to validate the correctness of the results. Various scale factors (SF) exist to help evaluate the scalability of the system.

Three queries require to find a shortest or cheapest path from some starting point to a target. These queries are 15 [15], 19 [16], and 20 [17]. Read query 20 [17] provides two parameters: `Person2` and `Company`. The goal is to find `Person1` working or has worked at `Compay`, who has a connection with `Person2`. `Person2` is not working or has not worked at `Company`. This connection is defined by persons who know each other and have studied at the same university. The weight of the connection is an integer of the absolute difference between the year of enrolment + 1 (as to avoid division by zero errors). Since the weights between any two persons are not all equal, it is necessary to calculate the cheapest path using the Batched Bellman-Ford algorithm discussed in Section 2.3. In case all weights were equal, MS-BFS, also discussed in Section 2.3 could be used.

For read query 19 [16] the parameters are City1 and City2. The goal is to find Person1 and Person2, who are from their respective city, where the interaction path is the cheapest. The interaction path is defined as the number direct reply Comments to a Message by the other Person. More interactions imply a cheaper path, calculated by 1/count(interactions), resulting in a 32-bit float. This means that the implementation of the cheapest path should work with both integers and floating point numbers.

Query 15 [15] involves a number of steps. At the start, two Persons are given as parameters. For these, the weighted shortest path needs to be calculated using the knows relation. The weight for any knows edge is calculated based on the interactions between the source and destination persons. Every direct reply from one person to a Post of the other person adds 1.0 interaction point. Every direct reply to a comment from one person to a Comment by the other person adds 0.5 interaction point. Only messages that are within a given [startDate, endDate] are considered. The edges are undirected, thus the interaction score is equal in both directions. The overall weight is calculated by $1/(\text{interaction score} + 1)$

Furthermore, we will be making use of query 13 of the interactive workload. This query can be used to evaluate the performance of the shortest path algorithm. For this query we are given two parameters Person1 and Person2, for which we have to find a shortest path through the knows table. If a path can be found, we should return the length of the path. If no path is found, we should return -1 as the distance. It can be the case that Person1 is the same as Person2, in that case we should return 0. The complexity in this query is the search space that is given. Unlike with the queries discussed from the BI workload, we cannot perform a filter on the number of edges in the graph. In this case, we have to potentially consider all knows edges that are provided in the dataset.

3 GOALS

The goal of this thesis is to build on the work done by Singh et al. described in Section 2.5. To further complete the integration of SQL/PGQ into DuckDB several UDFs need to be implemented. The next step after being able to compute the reachability will be to compute the minimal length of a path between any two nodes. The paths can either have a weight equal to 1, or a custom weight which is a positive number. In either case, the minimal path length can be computed using either Bellman-Ford or Dijkstra's algorithm. Once computing this is possible, we will work on retrieving all the nodes and edges contained in the shortest / cheapest path and return this in a query.

Another goal of this thesis is to identify and implement various optimizations in DuckDB especially related to graph queries. One of these optimizations has already been identified and considers the case when multiple identical hash tables are built due to duplicate join statements. These duplicate joins can occur in plain SQL queries, though are more common in SQL/PGQ queries due to the common occurrence of many-to-many relationships in graphs. For example, first joining a person table with a knows table (for person1_id), after which the person table is again joined with the knows table (for person2_id). In both instances, the same hash table is built for an equivalent join, resulting in equivalent hash tables, which causes redundant work. This is wasted computation time and storage, given that one hash table would suffice for both joins. The optimization will be to identify whenever these identical joins are done and ensure that only the minimally required amount of hash tables are built.

The research questions of this thesis will be:

(1) How can shortest and cheapest paths best be implemented using MS-BFS?
  (a) What are the trade-offs between using Bellman-Ford and Dijkstra's algorithms?
  (b) How can the number of states during computation be minimized to allow for optimal execution?
(2) What are the performance bottlenecks in SQL/PGQ?
  (a) How can these performance bottlenecks be experimentally evaluated?

4 METHODOLOGY

4.1 Shortest path

Computing the shortest path between two vertices in an unweighted graph can be done using MS-BFS. In the current implementation by Singh et al., it is possible to determine the reachability of a node. However, to determine the shortest path, some form of state needs to be kept, such that it can be tracked how many hops have been done before reaching the destination node. Often, graphs have the small-world property, the distance between any two nodes is very small compared to the size of the graph [54]. In addition, the shortest distance is always smaller than or equal to the diameter, the greatest distance between any two nodes, of the graph. Therefore, in unweighted graphs, the size of the values needed to be stored in memory can also be relatively small. For every source node for which the shortest distance is computed, it will likely be enough to store the distance within 8 bits. Since the distance can never be negative, an unsigned 8-bit integer would suffice, in the case that the diameter of the graph is less than 255. However, we should care not to exceed this limit as the integer will overflow, causing it to wrap around to 0. Whenever that would happen we should copy the values into 16-bit or 32-bit unsigned integers to ensure no overflow errors occur. This does reduce the efficiency of the MS-BFS algorithm compared to the reachability implementation. Computing the reachability only requires a 1-bit bool value per BFS, which is set to 1 in case the destination node is reachable, 0 otherwise. Using the AVX-512 operation, it is possible to execute 512 reachability searches at once. However, since computing the shortest path requires at least 8 bits to keep track of the distance, it will only be possible to execute $512/8 = 64$ BFS searches at a time. If we were to use 32-bit integers, only 16 BFS searches could be run. Unsigned integers are preferred over signed, since the length of a path can never be negative, there is no need for negative integers. Denoting that a node is unreachable can be done by using UINT_MAX [38]. During execution, we keep track of the depth of the MS-BFS. Whenever this depth reaches UINT_MAX, we know there is a chance for an overflow and make sure this is handled correctly.

To compute the cheapest path in weighted graphs, there are typically two algorithms that can be used, Dijkstra's algorithm using a Fibonacci heap and the Bellman-Ford algorithm. Then et al. [53] propose a batched Bellman-Ford algorithm to compute the geodesic distance in weighted graphs. They find that the performance of

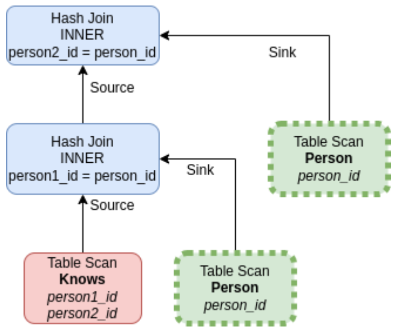

Figure 5: Example of a duplicate sink state in the physical plan

Month	Goals
February	Work on literature study Start on optimization shared hash tables
March	Work on optimization hash tables Start on shortest path using MS-BFS
April	Finish optimization hash tables Finish shortest path and experiment
May	Start on cheapest path using batched Bellman-Ford / Dijkstra
June	Finish cheapest path Start on experimenting
July	Finish up experimenting Finish up writing

Table 1: Initial planning

the Bellman-Ford algorithm is 3-10x higher compared to a batched variant of Dijkstra's algorithm [53]. Therefore, we will first look to implement the batched Bellman-Ford algorithm proposed by Then et al. as shown in Algorithm 3. Implementing a batched version of Dijkstra's algorithm will be more complex. During the batched version multiple instances of Dijkstra's algorithm will run, each having its Fibonacci heap. Given that each heap will be of a different structure, the effectiveness of possible SIMD instructions will be reduced. Therefore, it is not likely that a batched version of Dijkstra's algorithm will outperform the batched version of the Bellman-Ford algorithm. To test the performance of the implementations we will be using the LDBC Graphalytics benchmark [27], which is designed to test various algorithms such as single-source shortest path (SSSP) and BFS. Additionally, we will be using the LDBC Social Network Benchmark [4] to further test the performance of the implementations.

4.2 Shared hash join

Queries can contain multiple expensive joins which are in essence identical. This also happens in graph-like queries, where it is common to join the vertex table twice on the edge table. Once for the source of the edge, and once for the destination of the edge.

In the current state of DuckDB, for each join a hash table is built from the smaller table. However, this is wasteful in the case where queries containing multiple join operations have identical sinks. For example, there exists a vertex table *Person* containing the column `person_id`, and an edge table *Knows* containing the columns `person1_id` and `person2_id`. These tables are used to build a CSR representation for the edge table. Thus, it is necessary to perform two joins, namely `person1_id` ⋈ `person_id` and `person2_id` ⋈ `person_id`. In this case, the right side is identical in both joins. See Figure 5 for the relevant part of the physical plan of this example query. An optimization is to build this hash table only once, and reuse it for any identical joins containing the same sink. This will eliminate the need to build the same hash table multiple times.

The plan for the shared join optimization is to make the sink shareable between multiple pipelines in the case where the joins are identical. This will eliminate the need of building multiple identical hash tables since they can be reused. During the building phase of the pipelines, we will save all join operations we come across in combination with its respective pipeline. In the case where a duplicate join is encountered will the sink state of that pipeline be shared with the sink state of the pipeline seen earlier. In addition, the dependency of the parent pipeline on the current pipeline needs to be changed. The optimization can not be performed on FULL or RIGHT joins. For these joins a state needs to be kept for the hash tables to see if a particular value has been accessed or not. This access may vary between joins, and thus will result in different states. Therefore, the hash table cannot be reused for these types of joins.

In DuckDB, unit tests can be written using the *SQLLogicTests framework* to test the correctness of the implementation [31]. Assuming that the implementation works correctly, we will be testing the performance of the optimization using the CSR creation queries used by Singh et al. [48]. These queries contain a duplicate join on the vertex table. By performing tests with and without the optimization, we should see that the queries using the optimization have a lower run-time.

4.3 Planning

See Table 1.

5 IMPLEMENTATION

5.1 Shared hash join

The initial idea was to detect the duplicate joins in the optimizer. This can be done by adding an additional rule to the optimizer which scans for duplicate joins. In the optimizer, the logical plan tree would be traversed. Whenever a join is detected, the name of the table on which the join is performed is saved in the *client context*. The client context can be used to save variables to be used later in later phases during the query execution. The client context stays the same between queries, thus it is important to clean up whatever necessary of the client context at the end of every query. In case an identical table is detected multiple times in various joins in the same query, thus implying an identical join, we would replace the later encountered join with a special *SHARED HASH JOIN* logical node. This node would reference the left hand side table name of the original join, and a single child that represents the right hand side table used for the source.

There are a number of challenges with this approach. First of all, using only the name of the left hand side table to detect duplicate joins would not be sufficient. Multiple joins on the same table can exist that make use of different columns. These would then result in non-duplicate hash tables, which can therefore not be reused by each other. Thus it is important to also look at the conditions of the joins.

Assuming the duplicate join can be detected, a special logical node would have then replaced this join. However, this requires the introduction of a new logical node, and a new physical operator for the physical planner stage. This new physical operator would then have created its own pipeline. This would have created additional work that could be saved by using another approach that was ultimately chosen as the best suited option.

Instead of detecting the duplicate joins in the optimizer, they are detected during the pipeline creation. At this point, the physical plan has been created, and is being converted into pipelines as explained previously . When the physical operator *HASH JOIN* is detected, we save the corresponding pipeline and the *HASH JOIN* operator are saved in a dictionary inside the *Executor*, alongside all other pipelines. Then, whenever an additional *HASH JOIN* is detected, we check if this join is duplicate to those seen earlier. This is done by comparing the join type (INNER, OUTER, ANTI), the number of conditions, and ultimately if the right hand side of every condition is equal to the earlier detected join.

Insert ref to section here

check

5.2 Perfect join

In some cases it is possible to perform a *perfect join*. In DuckDB this is done when the table on which the hash table is built contains less than one million rows. In addition, there should not be too many conditions. Furthermore, it can only occur on an *INNER* or *LEFT* join.

6 RELATED WORK

6.1 Resource Description Framework

Another form in which graph data can be modelled, aside from the property graph model, is *Resource Description Framework* (RDF). Data is modelled in the form of triples, which is composed of a *subject*, *predicate*, and an *object* [46]. The nodes within an RDF can be of three types: Internationalised Resource Identifiers (IRIs) used to globally identify entities and relations, literals, and blank nodes [25]. Node attributes are modelled as an extra outgoing edge (being the predicate) to another node which is the object stored as a literal. One way of modelling edge attributes is described by Sun et al. [51], where every edge attribute requires four new edges being added to the graph. This method of modelling edge attributes is generally applicable, though verbose and inefficient in terms of storage [51]. Database management systems such as BlazeGraph [10] and Amazon Neptune [3] have based their data model on RDF. Storing RDF data in relational databases has some challenges [14]. One way of storing data is by using a single table with three columns: subject, predicate, object. However, large-scale RDF data runs into performance and scalability issues [39]. In particular, queries containing joins prove to be difficult to handle [39, 40].

6.2 Self-joins

Self-joins occur whenever a table joins with itself. For example, a knows table containing two columns (person1_id, person2_id) is joined in the following query:

```
SELECT knows1.person1_id, knows2.person2_id
FROM knows k1
JOIN knows k2 ON k1.person1_id = k2.person2_id;
```

Listing 7: Self-join on the knows table

As described by Szarnyas [52], there are several challenges with self-joins. An important one is related to RDFs that are stored as one table, which requires a join for every attribute queried [40]. Another challenge is related to the hash-joins that are often used by database systems to perform join operations. Whenever a join between two tables is performed, the smaller of the two tables is used to build a hash table. The other table is then used to probe the hash table and look for matches. However, with self-joins, both tables are equal in size, negating the advantage of building a hash table on a smaller table. In addition, the resulting table will probably be large in size when the table is of a many-to-many relation.

6.3 Mapping from Graph to Relational queries

Research has been conducted on the translation from a given query language to another one. Work that is particularly interesting to this thesis is translating graph query languages to SQL. An example of this is *Cytosm* (Cypher to SQL Mapping) by Steer et al. [50]. It acts as an application to execute graph queries on non-graph databases. In addition, they introduced *gTop*, a format that is able to capture the structure of property graphs and allow a mapping between graph query languages and a variety of database management systems. Steer et al. find that the translated SQL queries show comparable performance to GDBMSs [50].

Another example is GraphGen [59] that acts as an abstraction layer on top of an RDBMS. Underlying relational datasets are transformed and defined as graphs (*Graph-Views*). These graphs can then be queried using a graph API. The GraphGen framework has two main functionalities. First, users can define the structure of a graph using GraphGenDL, a Datalog-like domain-specific language (DSL). The other functionality is taking queries and executing them against the Graph-Views. The queries are specified by GraphGenQL, which is loosely based on SPARQL, Cypher, and PGQL [59].

SQLGraph allows Gremlin queries to be converted into SQL [51]. It uses a combination of relational and non-relational data storage. The *adjacency information* makes use of relational storage, while the attributes regarding all the nodes and edges are stored with JSON storage.

Lastly, there is IBM Db2 Graph by Tian et al. [55], which is an in-DBMS graph query approach that supports synergystic and retrofittable graph queries using the query language Gremin inside the IBM Db2 relational database. Similar to GraphGen, it has a *graph overlay* method that exposes *graph views* of the relational data.

Zhao and Yu showed that it was possible to support a large class of graph algorithms, such as BFS, Bellman-Ford, PageRank, in SQL [61]. They introduce new operations, alongside the existing ones (selection, projection, union, set difference, cartesian product, and rename) to provide explicit support for graph algorithms.

7 CONCLUSION

In this thesis we will be working on implementing the minimal path length as a UDF in DuckDB. This will be done using the batched Bellman-Ford algorithm, and/or Batched Dijkstra's algorithm developed by Then et al. The goal of this is to further complete the integrating of SQL/PGQ in DuckDB. Alongside, we will identify and implement optimizations that are specifically useful for graph-like queries. One of such optimizations is the shared hash join, which eliminates the need of building duplicate hash tables.

REFERENCES

[1] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 967–980. https://doi.org/10.1145/1376616.1376712

[2] Luís Amaral, Antonio Scala, Marc Barthelemy, and H. Stanley. 2000. Classes of Small-World Networks. *Proceedings of the National Academy of Sciences of the United States of America* 97 (11 2000), 11149–52. https://doi.org/10.1073/pnas.200327197

[3] Amazon. 2022. Amazon Neptune. https://aws.amazon.com/neptune/. Accessed: 02/02/2022.

[4] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020). arXiv:2001.02299 http://arxiv.org/abs/2001.02299

[5] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutiérrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1421–1432. https://doi.org/10.1145/3183713.3190654

[6] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40. https://doi.org/10.1145/3104031

[7] Michael J. Bannister and David Eppstein. 2011. Randomized Speedup of the Bellman-Ford Algorithm. arXiv:1111.5414 [cs.DS]

[8] Richard Bellman. 1958. On a routing problem. *Quarterly of applied mathematics* 16, 1 (1958), 87–90.

[9] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2019. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *CoRR* abs/1910.09017 (2019). arXiv:1910.09017 http://arxiv.org/abs/1910.09017

[10] BlazeGraph. 2022. BlazeGraphDB. https://blazegraph.com/. Accessed: 02/02/2022.

[11] Peter Boncz. 2022. The (sorry) State of Graph Database Systems. https://www.youtube.com/watch?v=aDoorU4X6Jk. Keynote at the EDBT/ICDT conference. Accessed: 04/04/2022.

[12] Peter Boncz, Renzo Angles, Oskar van Rest, Mingxi Wu, and Stefan Plantikow. 2022. A Survey Of Current Property Graph Query Languages. https://homepages.cwi.nl/~boncz/job/gql-survey.pdf. Accessed: 17/02/2022.

[13] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[14] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. 2013. Building an efficient RDF store over a relational database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 121–132. https://doi.org/10.1145/2463676.2463718

[15] Linked Data Benchmark Council. 2022. LDBC SNB BI read 15. https://ldbcouncil.org/ldbc_snb_docs/bi-read-15.pdf. Accessed: 13/06/2022.

[16] Linked Data Benchmark Council. 2022. LDBC SNB BI read 19. https://ldbcouncil.org/ldbc_snb_docs/bi-read-19.pdf. Accessed: 13/06/2022.

[17] Linked Data Benchmark Council. 2022. LDBC SNB BI read 20. https://ldbcouncil.org/ldbc_snb_docs/bi-read-20.pdf. Accessed: 13/06/2022.

[18] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Hannes Voigt, Oskar van Rest, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2021. Graph Pattern Matching in GQL and SQL/PGQ. *CoRR* abs/2112.06217 (2021). arXiv:2112.06217 https://arxiv.org/abs/2112.06217

[19] DuckDB Contributors. 2022. DuckDB Documentation - WITH Clause. https://duckdb.org/docs/sql/query_syntax/with. Accessed: 14/03/2022.

[20] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. 2015. The Case Against Specialized Graph Analytics Engines. In *CIDR*.

[21] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. *Proceedings of the 2018 International Conference on Management of Data* (2018). https://hal.archives-ouvertes.fr/hal-01803524/file/paper.pdf

[22] Michael L. Fredman and Robert Endre Tarjan. 1987. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. ACM* 34, 3 (jul 1987), 596–615. https://doi.org/10.1145/28869.28874

[23] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2009. *Database systems - the complete book (2. ed.)*. Pearson Education.

[24] Alastair Green. 2018. A proposal to the database industry: Not three but one: GQL. https://gql.today/wp-content/uploads/2018/05/a-proposal-to-the-database-industry-not-three-but-one-gql.pdf.

[25] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D'amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. 2021. Knowledge Graphs. *ACM Comput. Surv.* 54, 4, Article 71 (jul 2021), 37 pages. https://doi.org/10.1145/3447772

[26] Intel. 2022. Intel® AVX-512 Instructions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html. Accessed: 08/03/2022.

[27] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capota, Narayanan Sundaram, Michael J. Anderson, Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, and Peter A. Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.* 9, 13 (2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[28] Guodong Jin, Nafisa Anzum, and Semih Salihoglu. 2022. GrainDB: A Relational-core Graph-Relational DBMS. In *CIDR*.

[29] Laurens Kuiper. 2021. Fastest table sort in the West - Redesigning DuckDB's sort. https://duckdb.org/2021/08/27/external-sorting.html. Accessed: 04/05/2022.

[30] DuckDB Labs. 2021. New CWI spin-off company DuckDB Labs: Solutions for fast database analytics. https://duckdblabs.com/news/spin-off-company-DuckDB-Labs/. Accessed: 02/02/2022.

[31] DuckDB Labs. 2022. DuckDB - Testing. https://duckdb.org/dev/testing. Accessed 19/03/2022.

[32] Oracle Labs. 2022. Oracle PGX. https://docs.oracle.com/cd/E56133_01/latest/index.html. Accessed: 08/03/2022.

[33] ISO/IEC JTC 1/SC 32 Data management and interchange. [n.d.]. ISO/IEC CD 39075: Information Technology — Database Languages — GQL. https://www.iso.org/standard/76120.html. Accessed: 17/02/2022.

[34] ISO/IEC JTC 1/SC 32 Data management and interchange. [n.d.]. ISO/IEC CD 9075-16.2: Information technology — Database languages SQL — Part 16: SQL Property Graph Queries (SQL/PGQ). https://www.iso.org/standard/79473.html. Accessed: 17/02/2022.

[35] Jim Melton and Alan R. Simon. 2002. Sql: 1999 Understanding Relational Language Components.

[36] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: a large-scale subgraph query benchmark. In *GRADES-NDA '21: Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Virtual Event, China, 20 June 2021*, Vasiliki Kalavri and Nikolay Yakovets (Eds.). ACM, 8:1–8:11. https://doi.org/10.1145/3461837.3464516

[37] Jan Michels and Andy Witkowski. 2020. Graph database applications with SQL/PGQ. https://download.oracle.com/otndocs/products/spatial/pdf/AnD2020/AD_Develop_Graph_Apps_SQL_PGQ.pdf. Accessed: 08/02/2022.

[38] Microsoft. 2022. C and C++ Integer Limits. https://docs.microsoft.com/en-us/cpp/c-language/cpp-integer-limits?view=msvc-170. Accessed: 20/03/2022.

[39] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *VLDB J.* 19, 1 (2010), 91–113. https://doi.org/10.1007/s00778-009-0165-y

[40] M. Tamer Özsu. 2019. Graph Processing: A Panoramic View and Some Open Problems. In *VLDB*. https://vldb2019.github.io/files/VLDB19-keynote-1-slides.pdf

[41] Nous Populi. 2018. What is the most widely-used graph query language in 2018. https://leapgraph.com/graph-query-languages/. Accessed: 03/02/2022.

[42] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. https://doi.org/10.1145/3299869.3320212

[43] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph Databases, 2nd Edition.* O'Reilly Media, Inc.

[44] Marko Rogriguez, Yochai Benkler, and Hassan Masum. 2008. *A collectively generated model of the world. In: Collective intelligence: creating a prosperous world at peace.* Oakton: Earth Intelligence Network, 2008. 261–264 pages.

[45] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.* 29, 2-3 (2020), 595–618. https://doi.org/10.1007/s00778-019-00548-x

[46] Guus Schreiber and Yves Raimond. 2014. W3C RDF 1.1 Primer. https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/. Accessed: 11/03/2022.

[47] Bin Shao, Yatao Li, Haixun Wang, and Huanhuan Xia. 2017. Trinity Graph Engine and its Applications. *IEEE Data Eng. Bull.* 40, 3 (2017), 18–29. http://sites.computer.org/debull/A17sept/p18.pdf

[48] Tavneet Singh, Gábor Szárnyas, and Peter Boncz. 2022. Integrating SQL/PGQ to DuckDB. https://docs.google.com/presentation/d/1aeHCOuopl8r7Lk-TtqpYSEsXblni632rJoVUR62jc7s. Accessed: 03/02/2022.

[49] Neo4j Query Languages Standards and Research Team. 2018. GQL Scope and Features. (12 2018). https://s3.amazonaws.com/artifacts.opencypher.org/website/materials/sql-pg-2018-0046r3-GQL-Scope-and-Features.pdf

[50] Benjamin A. Steer, Alhamza Alnaimi, Marco A. B. F. G. Lotz, Félix Cuadrado, Luis M. Vaquero, and Joan Varvenne. 2017. Cytosm: Declarative Property Graph Queries Without Data Migration. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*, Peter A. Boncz and Josep Lluís Larriba-Pey (Eds.). ACM, 4:1–4:6. https://doi.org/10.1145/3078447.3078451

[51] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. 2015. SQLGraph: An Efficient Relational-Based Property Graph Store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1887–1901. https://doi.org/10.1145/2723372.2723732

[52] Gabor Szarnyas. 2022. Self-joins. https://szarnyasg.github.io/posts/self-joins/. Accessed: 11/03/2022.

[53] Manuel Then, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. 2017. Efficient Batched Distance, Closeness and Betweenness Centrality Computation in Unweighted and Weighted Graphs. *Datenbank-Spektrum* 17, 2 (2017), 169–182. https://doi.org/10.1007/s13222-017-0261-x

[54] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. 2014. The More the Merrier: Efficient Multi-Source Graph Traversal. *Proc. VLDB Endow.* 8, 4 (dec 2014), 449–460. https://doi.org/10.14778/2735496.2735507

[55] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Suijun Tong, Wen Sun, Thomas Kolanko, Md. Shahidul Haque Apu, and Huijuan Peng. 2020. IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 345–359. https://doi.org/10.1145/3318464.3386138

[56] TigerGraph. 2022. TigerGraphDB. https://www.tigergraph.com/. Accessed: 02/02/2022.

[57] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A Property Graph Query Language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems* (Redwood Shores, California) *(GRADES '16)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2960414.2960421

[58] Thomas Vilhena. 2019. Index-free adjacency. https://thomasvilhena.com/2019/08/index-free-adjacency. Accessed: 21/02/2022.

[59] Konstantinos Xirogiannopoulos, Virinchi Srinivas, and Amol Deshpande. 2017. GraphGen: Adaptive Graph Processing using Relational Databases. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*, Peter A. Boncz and Josep Lluís Larriba-Pey (Eds.). ACM, 9:1–9:7. https://doi.org/10.1145/3078447.3078456

[60] Jin Y. Yen. 1970. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quart. Appl. Math.* 27 (1970), 526–530.

[61] Kangfei Zhao and Jeffrey Xu Yu. 2017. Graph Processing in RDBMSs. *IEEE Data Eng. Bull.* 40, 3 (2017), 6–17. http://sites.computer.org/debull/A17sept/p6.pdf

A SQL VS SQL/PGQ QUERIES

```
WITH RECURSIVE paths(startPerson, endPerson, path) AS (
   SELECT -- define the path as the first edge of the
     traversal
        person1id AS startPerson,
        person2id AS endPerson,
        [person1id, person2id]::bigint[] AS path
     FROM knows
   UNION ALL
   SELECT -- concatenate new edge to the path
        paths.startPerson AS startPerson,
        person2id AS endPerson,
        array_append(path, person2id) AS path
     FROM paths
     JOIN knows ON paths.endPerson = knows.person1id
    WHERE knows.person2id != ALL(paths.path) -- detect
     cycles
)
SELECT startPerson, endPerson, path
FROM paths
WHERE startPerson = 1;
```

Listing 8: SQL query WITH RECURSIVE

```
select c1id, c2id, c3id
from GRAPH_TABLE (aml,
MATCH (c1 IS customer)-[IS transfers]->(c2 IS customer)-[
     t2 IS transfers]->(c3 IS customer)-[t3 IS transfers
     ]->(c1)
COLUMNS (c1.cid AS c1id, c2.cid AS c2id, c3.cid AS c3id)
     gt

/* The above SQL/PGQ query is transformed into the lower
     SQL query  */

select c1id, c2id, c3id
from (SELECT c1.cid as c1id, c2.cid as c2id, c3.cid as
     c3id
FROM customer c1, transfers t1, customer c2, transfers t2
   , customer c3, transfers t3
WHERE c1.cid = t1.from_id
    AND c2.cid = t1.to_id
    AND c2.cid = t2.from_id
    AND c3.cid = t2.to_id
    AND t3.from_id = c3.cid
    AND t3.to_id = c1.cid
)
```

Listing 9: SQL/PGQ query transformed into SQL query

```
select gt.c1id, gt.c2id
from GRAPH_TABLE (aml, MATCH (c1 IS CUSTOMER)-[t1 IS
     TRANSFERS*]->(c2 IS CUSTOMER) COLUMNS (c1.cid AS
     c1id, c2.cid AS c2id)) gt

/* The above SQL/PGQ query with a Kleene star is
     transformed into the lower SQL query  */
select c1id, c2id
FROM
 (
WITH cte1 AS (
SELECT min(CREATE_CSR_EDGE(0, (SELECT count(c.cid) as
     vcount FROM Customer c),
CAST ((SELECT sum(CREATE_CSR_VERTEX(0, (SELECT count(c.
     cid) as vcount FROM Customer c),
sub.dense_id , sub.cnt )) as numEdges
FROM (
    SELECT c.rowid as dense_id, count(t.from_id) as cnt
    FROM Customer c
    LEFT JOIN  Transfers t ON t.from_id = c.cid
    GROUP BY c.rowid
) sub) AS BIGINT),
src.rowid, dst.rowid ) ) as temp, (SELECT count(c.cid)
     FROM Customer c) as vcount
FROM
  Transfers t
  JOIN Customer src ON t.from_id = src.cid
  JOIN Customer dst ON t.to_id = dst.cid
)
SELECT src.cid AS c1id, dst.cid AS c2id
FROM cte1, Customer src, Customer dst
WHERE
( reachability(0, true, cte1.vcount, src.rowid, dst.rowid
    ) = cte1.temp)
);
```

Listing 10: SQL/PGQ query transformed into SQL query with Kleene star