Vrije Universiteit Amsterdam


Universiteit van Amsterdam

Master Thesis

# Integrating SQL/PGQ into DuckDB

**Author:** Daniël ten Wolde (2619049)

| | |
|---|---|
| *1st supervisor:* | Prof. Peter A. Boncz |
| *daily supervisor:* | Dr. Gábor Szárnyas (Centrum Wiskunde & Informatica) |
| *2nd reader:* | Prof. dr. ir. Henri E. Bal |

*A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science*

August 2, 2022

*"Luck is what happens when preparation meets opportunity."*
*by* Seneca

# Abstract

The upcoming SQL:2023 standard introduces the SQL/PGQ (Property Graph Queries) extension, which allows users (1) to define graph views over relational tables and (2) formulate graph pattern matching and path-finding operations using a concise syntax. These features allow efficient execution of graph queries within existing, proven relational database management systems (RDBMSs). However, as of 2022, SQL/PGQ lacks a reference implementation and research on integrating path-finding algorithms in RDBMSs has been limited.

In this thesis, we investigate the feasibility of integrating path-finding algorithms for shortest path problems in the open-source RDBMS DuckDB: unweighted shortest path, cheapest path, and any shortest path. Using a lightweight extension approach that relies on user-defined functions (UDFs), we adopted the multi-source breadth-first search and the batched Bellman-Ford algorithms to the vectorised execution model of DuckDB.

We evaluated the performance and scalability of our implementation using queries from the Linked Data Benchmark Council's Social Network Benchmark (LDBC SNB), which tests graph functionalities and performance of DBMSs. The algorithms implemented showed to scale linearly, and being able to handle the largest graphs currently available in the LDBC SNB. The results demonstrate that DuckDB is an ideal candidate to create an implementation of SQL/PGQ.

# Contents

**List of Figures** **v**

**List of Tables** **vii**

**1 Introduction** **1**

1.1 Context . . . . . 1

1.2 Research Questions . . . . . 4

1.3 Thesis structure . . . . . 4

**2 Background** **5**

2.1 Data Models . . . . . 5

2.1.1 Property Graph Data Model . . . . . 5

2.1.2 Resource Description Framework . . . . . 6

2.1.2.1 Self-joins . . . . . 7

2.2 Graph Query Languages . . . . . 8

2.2.1 Cypher . . . . . 8

2.2.2 G-CORE . . . . . 9

2.2.3 Property Graph Query Language (PGQL) . . . . . 9

2.2.4 SQL/Property Graph Queries . . . . . 10

2.3 Graph Traversal Algorithms . . . . . 13

2.3.1 Unweighted shortest path . . . . . 13

2.3.2 Cheapest path . . . . . 14

2.4 Single Instruction, Multiple Data Execution Model . . . . . 16

2.5 Cache latency . . . . . 20

**3 Related Work** **21**

3.1 Mapping Strategies . . . . . 21

3.1.1 Mapping from Graph to Relational Queries . . . . . 21

3.1.2 Mapping from RDF to PG . . . 22
3.2 Survey on Graph Database Management Systems . . . 23
3.3 Data processing systems supporting graph processing workloads . . . 24
3.3.1 GRADOOP . . . 24
3.3.2 The case against specialised graph analytics engines . . . 25
3.3.3 GRainDB . . . 25
3.3.4 MillenniumDB . . . 26

**4 Design & Implementation 29**
4.1 DuckDB . . . 30
4.1.1 DuckDB execution pipeline . . . 30
4.1.2 Join operator . . . 31
4.1.3 Design decisions . . . 32
4.1.4 Testing in DuckDB . . . 32
4.2 Current state of SQL/PGQ in DuckDB . . . 32
4.3 Shared hash join . . . 33
4.4 Compressed Sparse Row . . . 35
4.5 Shortest Path . . . 37
4.6 Any Shortest Path . . . 40
4.7 Cheapest Path . . . 46
4.7.1 Vectorising batched Bellman-Ford . . . 49

**5 Experiments 53**
5.1 Experimental Setup . . . 53
5.1.1 LDBC Social Network Benchmark . . . 53
5.2 Shortest Path . . . 56
5.3 Cheapest Path . . . 59
5.4 Vectorized batched Bellman-Ford . . . 60

**6 Discussion 63**
6.1 Performance comparison between shortest and cheapest path functions . . . 63
6.2 Vectorising batched Bellman-Ford . . . 65

**7 Future Work 67**
7.1 Path-finding optimisations . . . 67
7.2 CSR optimisations . . . 68
7.3 Miscellaneous improvements . . . 69

**8 Conclusion** **71**

**References** **73**

**A SQL vs SQL/PGQ queries** **85**

**B Any shortest path example** **87**

**C Literature study inclusion** **93**

# List of Figures


1.1 SQL/PGQ in relation to SQL and GQL . . . . . 3

2.1 Partial LDBC Social Network Benchmark schema depicted using a UML-like notation . . . . . 5
2.2 Example of a social network property graph . . . . . 7
2.3 Social network graph and where they study. . . . . 11
2.4 Scalar vs. Vectorised . . . . . 17
2.5 Comparison between Non-SIMD-ised and SIMD-ised instruction set. . . . . 18
2.6 Explicit SIMD code and the instructions generated by the compiler. . . . . 19

4.1 Schematic overview of key components for efficient path-finding operators . 30
4.2 Execution pipeline of DuckDB . . . . . 30
4.3 Example of a duplicate sink state in the physical plan . . . . . 34
4.4 Unweighted directed graph and the CSR representation . . . . . 36
4.5 Weighted directed graph and the CSR representation . . . . . 37
4.6 Initial state . . . . . 41
4.7 MS-BFS step 1 . . . . . 41
4.8 MS-BFS step 2 . . . . . 41
4.9 MS-BFS step 3 (final) . . . . . 42
4.10 Any shortest path initial state . . . . . 43
4.11 Any shortest path step 1 . . . . . 44
4.12 Any shortest path step 2 . . . . . 45

5.1 LDBC BI Query 19 . . . . . 54
5.2 LDBC BI Query 20 . . . . . 55
5.3 LDBC Interactive Query 13 . . . . . 55
5.4 Average execution time per scale factor for shortest path Interactive query 13 56

## LIST OF FIGURES

5.5 Average execution time per source-destination pair per scale factor for shortest path Interactive query 13 . . . . . 57
5.6 Relative time spent per phase for Interactive query 13 shortest path . . . . 58
5.7 CSR creation time for all scale factors . . . . . 58
5.8 CSR creation time for the scale factors greater than or equal to 100 . . . . . 59
5.9 Total execution time per scale factor for cheapest path Interactive query 13 59
5.10 Average execution time per source-destination pair per scale factor for cheapest path Interactive query 13 . . . . . 60
5.11 Relative time spent per phase for Interactive query 13 cheapest path . . . . 61
5.12 Performance of scalar vs. auto-vectorized implementations of batched Bellman-Ford . . . . . 61

6.1 Difference in performance between shortest and cheapest path for query 13 . 63

B.1 Any shortest path initial state . . . . . 88
B.2 Any shortest path step 1 . . . . . 89
B.3 Any shortest path step 2 . . . . . 90
B.4 Step 3 . . . . . 91
B.5 Any shortest path step 4 / final state . . . . . 92

# List of Tables

2.1 Variatons of a graph-traversal algorithm for various scenarios . . . . . 13
2.2 Event with corresponding latency, also scaled to 1 second . . . . . 20

## LIST OF TABLES

# 1

# Introduction

## 1.1 Context

There is a growing desire to perform more complex analyses on the increasing amounts of data being gathered. A significant value of large data sets is that they capture *connections* between their entities. It is intuitive to represent and think of these connections as *graphs* (1) A comprehensive survey of Sahu et al. (2) shows that graphs are used across various domains. Graphs often provide a natural way to structure data involving entities, represented as vertices, and the connections (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) (3).

These systems provide a graph data model, a graph query language, and have built-in graph visualisation capabilities. However, the performance of these systems often leaves to be desired (3, 4), which hampers their adoption. Meanwhile, 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 due to the limitations of the standard query language SQL (5).

In response to the limited capabilities, a plethora of GDBMSs arose in the last fifteen years, each having its graph query language (6). Examples of systems and their query languages are TigerGraph with GSQL (7), Neo4j with Cypher (8), and Oracle Labs PGX with PGQL (9). Amazon Neptune even supports three distinct languages: OpenCypher, Gremlin, and SPARQL. Using 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, semantics, and capabilities. This exposes the user to the threat of

vendor lock-in. The combination of 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 June of 2023 called SQL/Property Graph Queries (SQL/PGQ), making it easier to support this type of workload related to graph data in RDBMSs (10).

For querying graph data, two functionalities are deemed most important: *graph pattern matching* and *path-finding* (11). 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 (12). In particular, path-finding requires using recursive queries. An example can be seen in Listing 1.1, which shows a path-finding query that returns the count of persons living in the city of Delft, who can be reached through a transitive `follows` connection starting from the person named 'Daniel' (13). In contrast, the same query in SQL/PGQ is shown in Listing 1.2.

```
WITH RECURSIVE paths(startNode, endNode, path) AS (
    SELECT -- Initialisation
        p1id AS startNode,
        p2id AS endNode,
        [p1id, p2id] AS path
      FROM follows
    UNION ALL
    SELECT -- Recursion
        paths.startNode AS startNode,
        p2id AS endNode,
        array_append(path, p2id) AS path
      FROM paths
      JOIN follows ON paths.endNode = p1id
      WHERE p2id != ALL(paths.path)
    )
SELECT count(p2.id) AS cp2
FROM person p1
JOIN paths     ON paths.startNode = p1.id
JOIN person p2 ON p2.id = paths.endNode
JOIN city      ON city.id = p2.livesIn AND city.name = 'Delft'
WHERE p1.name = 'Daniel';
```

**Listing 1.1:** SQL query using `WITH RECURSIVE`

```
SELECT count(gt.id) AS cp2
FROM GRAPH_TABLE (socialNetwork,
  MATCH
   (p1:person WHERE name = 'Daniel')-[:follows]->+ -- Kleene plus
   (p2:person)-[:livesIn]->(c:city WHERE name = 'Delft')
  COLUMNS (p2.id)
) gt
```

**Listing 1.2:** SQL/PGQ query equivalent to Listing 1.1

Support for the `RECURSIVE` statement was added in SQL:1999 (14), and is supported by popular RDBMSs such as PostgreSQL, MySQL, and SQLite. However, as shown by Michels and Witkowski (5), even relatively simple graph queries in plain SQL take up many lines and are more difficult to understand than 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 more accessible.

Work on a standardisation of a graph query language started in 2018 (15, 16, 17). 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 (10).

We have received access to the SQL/PGQ specification through LDBC's liaison with ISO. In SQL/PGQ, a graph can be defined in terms of tables (18) and queries can contain special syntax for path-finding and graph pattern matching. The standard is limited to read-only queries, making it impossible to modify the graph through the graph tables in SQL/PGQ. Therefore, the same workgroup is also working on Graph Query Language (GQL) (19), in which it will be possible to modify the data in addition to most features in SQL/PGQ, see Figure 1.1. GQL and SQL/PGQ will share the same pattern matching syntax (16).


**Figure 1.1:** SQL/PGQ in relation to SQL and GQL

In this thesis, we integrate parts of the SQL/PGQ standard in DuckDB (20). DuckDB is an open-source in-process SQL OLAP database management system originating from CWI (21). With the new SQL/PGQ standard releasing soon, work on integrating it into DuckDB has already started by Singh et al. (22). However, the standard has not been fully integrated as of now. In particular, certain path-finding algorithms have not been implemented yet. Two cases exist for path-finding. One where the graph is unweighted, meaning the edges do not have a weight assigned. In this case, the shortest path function

will be implemented. The cheapest path function will be implemented in the other case where the graph is weighted. The feature to return the nodes contained in these cheapest and shortest path functions will also be implemented. Therefore, the goal is to further complete this integration by adding these functionalities. Finally, we will identify and implement optimisations primarily related to typical SQL/PGQ queries expected to be ran in DuckDB.

## 1.2 Research Questions

The following research questions have been defined for this thesis work.

1. How to best implement path-finding algorithms in DuckDB?
   (a) How can path-finding best be implemented for unweighted graphs?
   (b) How can path-finding best be implemented for weighted graphs?
2. What are the bottlenecks in the current SQL/PGQ implementation?
   (a) How can these bottlenecks be optimised?
   (b) What is the performance impact of vectorisation on the path-finding algorithms?

## 1.3 Thesis structure

The remainder of this thesis is structured as follows. Chapter 2 will provide information on data models used to represent graphs, existing graph query languages, the upcoming SQL/PGQ standard, and various path-finding algorithms. Chapter 3 shows related work and comments on it. Chapter 4 discusses the most important design and implementation details. It introduces DuckDB and how the various path-finding algorithms and optimisations have been implemented. Chapter 5 shows the results of the experiments conducted. Chapter 6 provides an in-depth discussion on the results obtained. Finally, chapter 7 and chapter 8 discuss future work and provide a conclusion to the thesis.

# 2

# Background

As our running example, we use a small property graph instance that conforms to the Linked Data Benchmark Council (LDBC) Social Network Benchmark's (SNB) schema as shown in Figure 2.1.

## 2.1 Data Models

### 2.1.1 Property Graph Data Model

Graphs can model complex, connected data through vertices and edges. The vertices, sometimes referred to as nodes, represent objects. The edges, also referred to as relationships, represent the connections (relations) between objects. The simplest form of a graph is the *simple directed graph model* (6). 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

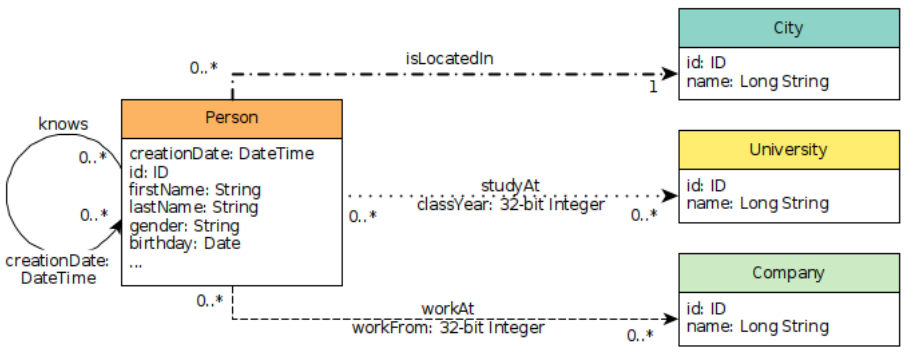



**Figure 2.1:** Partial LDBC Social Network Benchmark schema depicted using a UML-like notation

destination vertex. Graphs can either be weighted or unweighted. In weighted graphs, an edge between two vertices is assigned a weight. This weight can be different for every edge. In unweighted graphs, all edges are of equal weight.

The simple graph and weighted graph models suffice in some instances, such as computing the reachability of a vertex from all other vertices or calculating the cheapest paths. However, storing any information in either the vertices or edges is impossible. For this the *Labelled Property Graph (LPG)* model was introduced, which is often used in GDBMSs (6).

The LPG model enriches the simple graph model with the ability to assign labels and properties to vertices and edges. An example can be seen in Figure 2.2. In this case the vertices can have the labels *Person*, *City*, or *University*. There is also the possibility of assigning multiple labels to a single vertex. For clarity's sake, it was chosen not to do this in Figure 2.2.

In the property graph data model, it is possible to assign labels to edges, just like vertices. In Figure 2.2 there are various edge labels. First, there is the *knows* label for which the source vertex is of label *Person*, and the destination vertex is also of label *Person*. Second there is *studyAt* which starts from a *Person* and points to a *University*. Finally, there is *locatedIn* which goes from a *Person* and ends at a *City*.

In addition to labels, it is also possible for vertices and edges to have properties (attributes) typically related to the labels. These properties contain more specific information about the given vertex or edge. An example can be seen in Figure 2.2 where every *Person* vertex has the property *firstName*, providing more specific information on the vertex.

Examples of database systems that have based their data model on the LPG model are Neo4j (23), TigerGraph (7), and Oracle PGX (24).

### 2.1.2 Resource Description Framework

The *Resource Description Framework* (RDF) is a standard data model used for data interchange on the Web (25). Data is modelled in the form of *triples*, which is composed of a *subject*, *predicate*, and an *object* (25). The nodes within RDF can be of three types: Internationalised Resource Identifiers (IRIs) used to globally identify entities and relations, literals, and blank nodes (26). A node attribute is modelled as an extra outgoing edge (being the predicate) to another node which is the object stored as a literal. A way of modelling edge attributes is described by Sun et al. (27), where every edge attribute requires four new edges to be added to the graph. This method of modelling edge attributes is generally applicable, though verbose and inefficient in terms of storage (27). Database

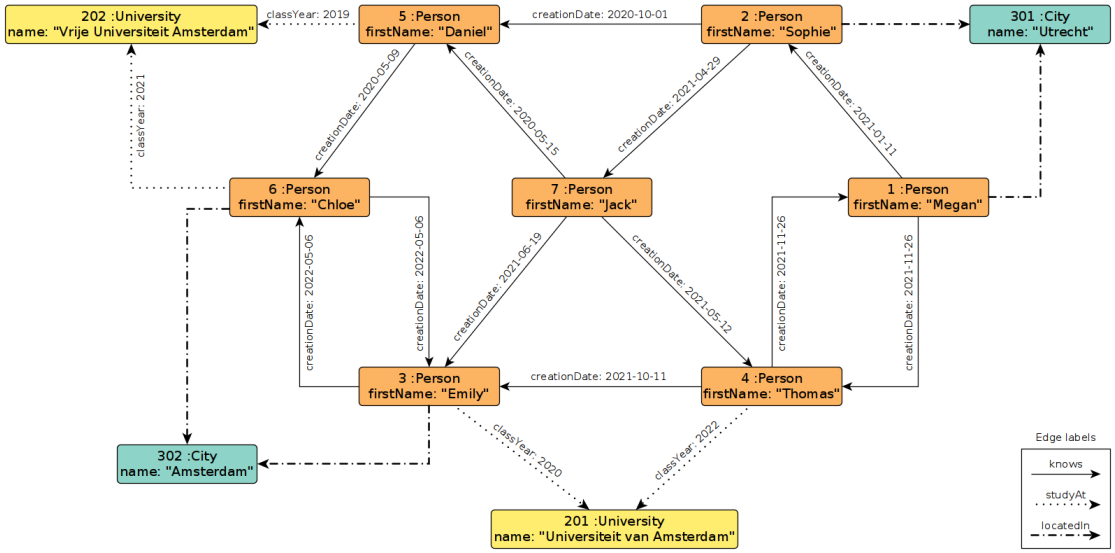



**Figure 2.2:** Example of a social network property graph

management systems such as BlazeGraph (28) and Amazon Neptune (29) have based their data model on RDF.

Storing RDF data in relational databases comes with several challenges (30). One of the challenges is how to store the data. A naïve approach uses a single table with three columns: subject, predicate, and object. However, large-scale RDF data runs into performance and scalability issues (31). In particular, queries containing joins prove difficult to handle (4, 31).

#### 2.1.2.1 Self-joins

A challenge with storing the data in a single table is related to self-joins, i.e. joins between instances of the same table. In this case, a join is required for every attribute queried to be retrieved from the table. For example, a *knows* table containing two columns (person1_id, person2_id) is joined in the following query:

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

**Listing 2.1:** Self-join on the knows table

Szárnyas describes several challenges related to self-joins (32). One of the challenges is related to the hash-joins often used by database systems to do 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 significant in size when the table is of a many-to-many relation.

## 2.2 Graph Query Languages

Up until now, there has not been a standardised graph query language. For RDBMSs, there exists SQL. However, SQL is arguably not as convenient to express graph workload queries, which often involve recursive joins (12). To query graph data, two functionalities are deemed most important: *graph pattern matching* and *path-finding* (11). Specialised graph query languages, such as Cypher and PGQL, have introduced specialised syntax that has made it easier to express queries involving these functionalities. This section provides an overview of the current most widely used graph query languages and describes SQL/PGQ and GQL.

### 2.2.1 Cypher

Francis et al. (8) describe Cypher, a language for querying and updating property graph databases and formalise its read-only core. The language started in Neo4j (33), a property graph database, but has since also been implemented in other industrial products such as SAP HANA Graph (34) and Redis Graph (35). It has also been used for academic research. For example, Sarma et al. used Cypher and Neo4j to detect bank fraud (36). Another example is Cytosm by Steer et al., which converts Cypher queries into plain SQL queries (37), see Section 3.1. Cypher is based on the LPG data model, just like G-CORE (15), see Section 2.2.2 and PGQL (9), see Section 2.2.3.

Queries are structured linearly in Cypher. According to the authors, this allows users to think of query processing as starting from the beginning and progressing linearly to the end (8). Whereas SQL queries start with a `SELECT` statement, Cypher queries end with a `RETURN` statement. An example of a Cypher query can be seen in Listing 2.2. In this query we wish to bind `person1` with the vertices labelled *Person* who have the *firstName* property "Thomas". From this we follow all *knows* edges from person1 and return the information related to `person2`. The equivalent SQL query can be seen in Listing 2.3.

```
MATCH
    (person1:Person {firstName: 'Thomas'})-[:KNOWS]-(person2:Person)
RETURN person2
```

**Listing 2.2:** Simple matching example in Cypher

```
SELECT p2.*
FROM person p
JOIN knows k on k.source = p.id
JOIN person p2 on k.destination = p2.id
WHERE p.firstName = "Thomas";
```

**Listing 2.3:** Equivalent matching query in SQL

A key feature of Cypher is the "ASCII art", also referred to as visual graph syntax and pattern matching style. The visual graph syntax way of describing a pattern is also seen in SQL/PGQ and GQL (16), see Section 2.2.4. Data modification is also possible in Cypher (38), using basic clauses such as `CREATE` to create new nodes and relationships, `DELETE` for removing entities, `SET` for updating properties and `MERGE` which tries to match a pattern, and creates one if it is not found in the database.

### 2.2.2 G-CORE

G-CORE is a combined effort of academia and industry to construct the future of graph query languages (15). Angles et al. identify three main challenges with existing graph query languages. Firstly, current implementations output tables of values, nodes, or edges. Instead, the output should be graphs to improve the usability, also referred to as composability. Secondly, G-CORE claims to be unique in the way paths are treated as first-class citizens. The authors argue that languages should be able to return paths to the user to enable post-processing paths within the query language instead of in an ad-hoc manner. The authors emphasise that G-CORE is a *design language* meant to guide other (future) query languages towards making them more practical, powerful, and expressive. Thus, it is essential to capture the core of available languages and take what they do well to create a standard. Therefore, the final challenge is related to the fact that no standard exists at the time of publishing the G-CORE paper in 2018.

### 2.2.3 Property Graph Query Language (PGQL)

Van Rest et al. identify that graph-based approaches to data analysis have become more popular over the last couple of years (9). According to the authors, the standard way of querying this data using SQL does not sufficiently cover all the graph-based approach

functionalities. They propose a new query language, PGQL with additional features include reachability analysis, path-finding and graph construction. The language is based on the property graph data model. The syntax of PGQL is SQL-like, which makes it intuitive to use for SQL users. PGQL support two different pattern matching semantics: isomorphism and homomorphism. With isomorphism, two entities are not allowed to map to the same node or edge. With homomorphism, this is allowed to happen. PGQL has vertex, edge, path, and graph types. The last one allows for the support of graph transformation applications, which allows one or more graphs to be constructed and returned from a query (9).

### 2.2.4 SQL/Property Graph Queries

Providing a way to perform graph processing analyses has been difficult due to the limitations of the standard query language SQL (5). The two most crucial graph querying functionalities are graph pattern matching and path-finding as described by Angles et al. (11). These functionalities become more accessible with the addition of SQL/PGQ, and queries involving these can be more easily expressed (5, 12).

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 (39). A graph can be defined using the SQL statement (40) found in Listing 2.4.

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

**Listing 2.4:** Creating a graph in SQL/PGQ

For example, if we wish to create the schema of the graph of Figure 2.3, symbolising a group of friends, some of whom study at a university, we would use the following query:

```
1 CREATE PROPERTY GRAPH social_network
2 VERTEX TABLES (
3     Person PROPERTIES ( personId, firstName ),
4     University PROPERTIES ( universityId, name )
5 )
6 EDGE TABLES (
7     knows SOURCE Person DESTINATION Person PROPERTIES ( creationDate ),
8     studyAt SOURCE Person DESTINATION University PROPERTIES ( classYear )
9 )
```

**Listing 2.5:** Creating a social network graph in SQL/PGQ

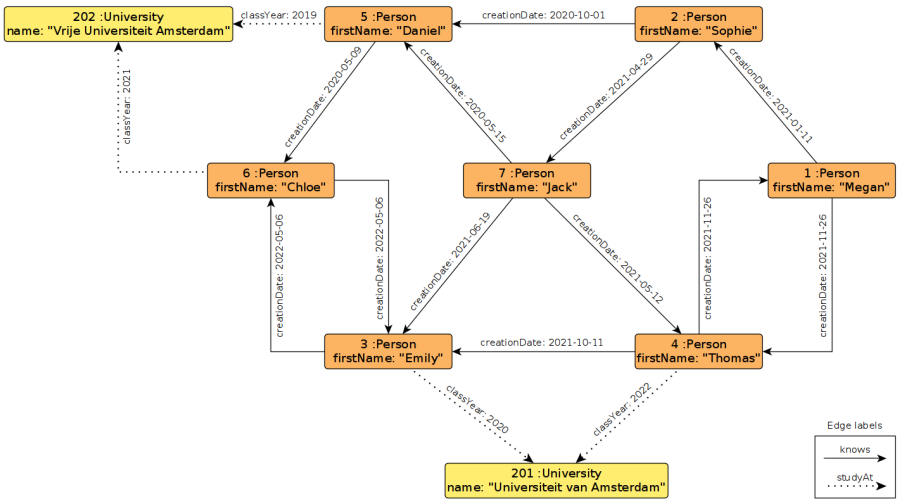



**Figure 2.3:** Social network graph and where they study.

To match a pattern to this graph in SQL/PGQ, the `MATCH` syntax can be used (16), as can be seen in Listing 2.6. For example, Listing 2.6 selects the personid and first name of persons from the graph in Figure 2.3 whose name is equal to 'Daniel'.

```
1 SELECT p.personId, p.firstName
2 FROM Person GRAPH_TABLE ( social_network,
3     MATCH ( a:Person WHERE a.firstName = 'Daniel' )-[ :studyAt ]->( u:
    University )
4     COLUMNS (
5         a.personId,
6         a.firstName )
7     ) p
```

**Listing 2.6:** Pattern matching all nodes with the property firstName Daniel who study at a university

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 its syntax makes it obvious whether a variable refers to a vertex or an edge. As can be seen in Listing 2.6, we use the `()` notation to denote a node. Denoting an edge can be done using `[]`.

To indicate that an edge is pointing from source to destination we write `(source)-[` *spec* `]->(destination)` (16). The arrow is an example of a directed edge pattern, which in this case, points right. However, it is also possible to point left, or be undirected. The visual graph syntax is inspired by the syntax of Cypher; see Section 2.2.1.

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

```
MATCH ( a:Person WHERE a.name = 'Jack' )-[ x:knows ]-( b:Person )
  -[ y:studyAt ]->( c:University )
```

**Listing 2.7:** Pattern matching using nodes and edges

This statement would extract all patterns that match vertex *a* being a Person with the name "Jack", who knows a person studying at a university. Within every node or edge, we can filter the possibilities by adding a `WHERE` statement, as can be seen in Listing 2.7.

One of the features of SQL/PGQ is the ability to match a single edge pattern or a parenthesised path pattern for an arbitrary length (16). 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 2.8:** Path length of 2 to 5 knows edges

It can be argued that finding such a path in plain SQL is more difficult to express as it requires the union of multiple joins on the `knows` table (5). 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 one 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 2.9:** Path length of arbitrarily many knows edges

An example of a Regular Path Query (RPQ) possible in SQL/PGQ can be seen in Listing 2.10.

```
MATCH ( a:Person )-
    [ -[ e1:knows WHERE creationDate > 2019-01-01]->
        (c:Person)
        -[ e2:knows WHERE creationDate > 2020-01-01]-> ]{2,5}->
    ( b:Person )
```

**Listing 2.10:** Repeating multiple edges

Queries in SQL/PGQ can also perform path-finding operations. It will be possible to specify that any shortest path matching the graph pattern should be returned using the `ANY SHORTEST` keyword. This keyword is non-deterministic as potentially multiple paths

| | Shortest | Cheapest |
|---|---|---|
| Single source<br>Single destination | Bidirectional BFS | Bidirectional Dijkstra |
| Single source<br>All destination | BFS | Dijkstra<br>Bellman-Ford |
| Multi-source<br>All destination | Multi-source BFS | Multi-source Bellman-Ford<br>Multi-source Dijkstra |
| All source<br>All destination | All-source BFS | Floyd-Warshall |

**Table 2.1:** Variatons of a graph-traversal algorithm for various scenarios

of equal length correspond to the same pattern. In this case, any of the matched patterns can be returned. Another option is to return all shortest paths, which is deterministic. Finally, it is possible to return the shortest $k$ paths, which is also a non-deterministic operation. Another addition in SQL/PGQ is the ability to return the nodes found on a path corresponding to a query.

Performing path-finding operations on a weighted graph is currently marked as a *language opportunity* and is likely to be included in a later version of SQL/PGQ.

## 2.3 Graph Traversal Algorithms

Multiple variations exist for graph traversals. Table 2.1 shows an overview of some algorithms that can be used for path-finding in various scenarios.

### 2.3.1 Unweighted shortest path

With unweighted shortest path, the goal is to find the path between a source vertex and a destination vertex with the minimal amount of intermediate vertices. The shortest path for unweighted graphs can be computed using the breadth-first search algorithm. This algorithm uses a search tree data structure, which does not contain cycles. However, the algorithm can be extended to graphs by preventing a vertex from being explored if it has been seen before.

The exploration starts from a single vertex and explores all its direct neighbours. Once all neighbours of the source vertex are explored, the subsequent iterations explore the neighbours of the neighbours. Thus, the vertices with an equal distance to the source vertex are discovered in the same iteration. The worst-case time complexity of the breadth-first

search is $O(|V| + |E|)$, indicating that the time scales linearly to the number of vertices and edges in the graph.

**Algorithm 1** Breadth-first search

```
    Function BFS(Graph, source)
2:      let Q be a queue
        label source as explored
4:      Q.enqueue(source)
        while Q is not empty do
6:          v := Q.dequeue()
            if v is the goal then
8:              return v
            end if
10:         for each edge from v to w in G.neighbours(v) do
                if w is not labelled as explored then
12:                 label w as explored
                    Q.enqueue(w)
14:             end if
            end for
16:     end while
```

### 2.3.2 Cheapest path

With cheapest path, commonly referred to as the weighted shortest path, the goal is to find the path with the minimal edge weight cost from a source vertex to a destination vertex. To compute the cheapest path in weighted graphs the two textbook algorithms used are Dijkstra's algorithm and Bellman-Ford. In both of these algorithms, the graph is explored from a single source. However, they differ in how the graph is explored.

In Dijkstra's algorithm, see Algorithm 2, a graph and a single source vertex is given. The vertices in the graph are split into two sets. One set contains all vertices in the shortest path tree, and the other set contains all other vertices that have not been explored yet. The shortest path tree is then incrementally generated with the source vertex as the root. In every step of the algorithm, a vertex from the not yet included set with the minimal distance from the source vertex is added to the shortest path tree. This can be seen as a priority queue where the vertex with the lowest distance gets chosen. Thus, as soon as a vertex is added to the cheapest path tree, the minimal distance from the source to this vertex is known. Dijkstra's algorithm has the constraint that there can only be

**Algorithm 2** Dijkstra's algorithm

```
    Function Dijkstra(Graph, source)
 2:     for each v in Graph do
            dist[v] := infinity
 4:         previous[v] := undefined
        end for
 6:     dist[source] := source
        Q := Set of all nodes in Graph
 8:     while Q is not empty do
            u := node in Q with smallest dist
10:         Remove u from Q
            for each neighbour v of u do
12:             alt := dist[u] + weight(u, v)
                if alt < dist[v] then
14:                 dist[v] := alt
                    previous[v] := u
16:             end if
            end for
18:     end while
```

non-negative edge weights. An optimisation for Dijkstra's algorithm is to implement the priority queue as a Fibonacci heap.

Bellman-Ford, shown in Algorithm 3 is an alternative algorithm to compute the cheapest distance and does not have the constraint that there must only be non-negative edge weights. In Bellman-Ford, a graph and a single source vertex is given. The distances from this source to all other vertices are stored in an array. Upon initialisation, the distance for the source vertex is 0. All other vertices have a distance of $\infty$ since it is not yet known how these vertices can be reached.

The algorithm iterates over all vertices in order and the edges are traversed for every vertex. If the distance of the current vertex plus the edge weight from the current vertex to its neighbour is less than the current known cheapest distance to its neighbour, a cheaper path is found and the distance is updated.

If all vertices have been iterated over without updating the distance array, all cheapest paths have been found, and the algorithm terminates. At most, $|V| - 1$ iterations over all vertices need to be made, where $|V|$ is the number of vertices. In every iteration, longer cheapest paths are found and only at the last iterations is the cheapest path guaranteed

**Algorithm 3** Bellman-Ford

```
Function Bellman-Ford(Graph, source)
    for i ← 1 to |V[G]| − 1 do
        for each edge (u, v) ∈ E[G] do
            Relax(u, v, w)
        end for
    end for
    for each edge(u, v) ∈ E[G] do
        if d[v] > d[u] + w(u, v) then
            Return False
        end if
    end for
```

to be found.

Dijkstra's algorithm using a Fibonacci heap has a worst-case time complexity of $\mathcal{O}(|E| + |V| \log |V|)$ (41), which is better than Bellman-Ford's worst-case time complexity of $\mathcal{O}(|V| \cdot |E|)$ (42). However, the expected runtime of Bellman-Ford is $O(|E|)$ in large dense graphs with low diameter (43).

Both Dijkstra's algorithm and Bellman-Ford are single source algorithms. Floyd-Warshall, see Algorithm 4, can be used to calculate the cheapest distance for all pairs of vertices. The restriction is that there can not be negative cycles in the graph; however, edges with negative weights are allowed. The same restriction holds for Bellman-Ford.

Floyd-Warshall uses a matrix with the dimensions $|V| \cdot |V|$ where $|V|$ is the number of vertices in the graph. During initialisation, the distances where source = destination are set to zero. The distances to the direct neighbour for every $v$ are set. The algorithm then loops over all possible combinations of vertices and checks for a cheaper distance. The worst-case time complexity, $O(V^3)$, equals the best-time complexity.

## 2.4 Single Instruction, Multiple Data Execution Model

In order to increase the CPU efficiency on the aformentioned algorithms, a potential optimization is to make use of Single Instruction, Multiple Data (SIMD) instructions. As the name suggests, SIMD allows a single instruction to be performed on multiple data.

SIMD instructions are helpful in a loop, where scalar instructions would execute the exact instructions for every iteration in the loop. With SIMD instructions, we optimise out the loop by using only a single instruction on multiple pieces of data, see Figure 2.4.

**Algorithm 4** Floyd-Warshall

```
Function Floyd-Warshall(Graph)
    |V| := number of vertices in the graph
    dist := |V| × |V| array of minimum distances
    for each vertex v do
        dist[v][v] := 0
    end for
    for each edge(u,v) do
        dist[u][v] = weight(u,v)
    end for
    for k from 1 to V do
        for i from 1 to V do
            for j from 1 to V do
                if dist[i][j] > dist[i][k] + dist[k][j] then
                    dist[i][j] := dist[i][k] + dist[k][j]
                end if
            end for
        end for
    end for
```

The requirement is that the data is aligned, meaning that all individual pieces of data are of the same type, such as an 8-bit integer.

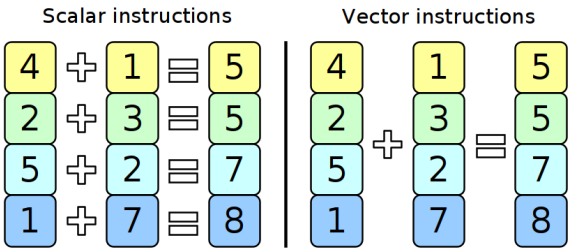



**Figure 2.4:** Scalar vs. Vectorised

An example can be observed in Figure 2.5a [1], where we have two arrays $A$ and $B$, each containing 16 8-bit integers. We wish to add the values of these two arrays and assign the value to array $C$. When adding these values using scalar operations, we require 16 addition instructions, which can be seen in Figure 2.5b. The instruction `movzx` copies the content of the source (`BYTE PTR [rsp+16+rax]` in this case, to the destination (`edx`). Then

[1] See `https://godbolt.org/z/d6v319sEq` for full instructions.

an `add` instruction is executed, which adds the destination operand (`dl`) and the source operand (`BYTE PTR [rsp+32+rax]`) and stores the result in the destination operand. The result is then moved using the `mov` instruction. The five instructions following are related to the for-loop. The `jne` instruction ensures that as long as iterator $i$ is not equal to 16 (checked with the `cmp` instruction), a jump is made back to the start of the instructions for the for-loop. Thus, only in the last iteration do we execute the final two instructions. The total number of instructions per iteration of the for-loop is six, except for the last iteration, which has 8 instructions. Therefore, when executing this for-loop, we require roughly $6 \cdot 15 + 8 = 98$ instructions.

```
for (int i = 0; i < 16; i++) {
    C[i] = A[i] + B[i];
}
```

**(a)** A for-loop to add two numbers in an array

```
.L4:
    movzx   edx, BYTE PTR [rsp+16+rax]
    add     dl, BYTE PTR [rsp+32+rax]
    mov     BYTE PTR [rsp+rax], dl
    add     rax, 1
    cmp     rax, 16
    jne     .L4
    mov     rbx, rsp
    lea     rbp, [rsp+16]
```

**(b)** Non-SIMD-ised instruction set

```
movdqa  xmm1, XMMWORD PTR .LC1[rip]
movdqa  xmm0, XMMWORD PTR .LC2[rip]
lea     rbx, [rsp+32]
lea     rbp, [rsp+48]
movaps  XMMWORD PTR [rsp+16], xmm0
paddb   xmm0, xmm1
movaps  XMMWORD PTR [rsp], xmm1
movaps  XMMWORD PTR [rsp+32], xmm0
```

**(c)** SIMD-ised instruction set

**Figure 2.5:** Comparison between Non-SIMD-ised and SIMD-ised instruction set.

On the other hand, Figure 2.5c shows that adding these two arrays can be performed in just three instructions using special SIMD registers. The data is loaded into the registers `xmm0` and `xmm1` using the `movdqa` instruction. When the values are loaded into the registers, they can be added using the `paddb` instruction. This instruction performs a SIMD add operation, similar to the `add` operation, adding the source and destination operands and storing the result in the destination operand. The values from registers `xmm1` and `xmm0` are then moved to a memory location.

Various versions of SIMD instructions exist, with the oldest being the SSE instruction set introduced initially with the Intel Pentium III in 1999 (44). These came with the XMM registers seen in the instructions in Figure 2.5c. The XMM registers are 128-bits wide.

SSE2 was introduced with the Intel Pentium 4 in 2000 (45). It added the support for the double-precision data type, allowing operations to be performed on the full range of

```
__m128i A_vec = _mm_load_si128((__m128i*)&A);
__m128i B_vec = _mm_load_si128((__m128i*)&B);
__m128i C_vec = _mm_add_epi8(A_vec,B_vec);
_mm_store_si128((__m128i*)&C,C_vec);
```

**(a)** Explicit SIMD for adding two arrays

```
movdqa  xmm1, XMMWORD PTR .LC1[rip]
movdqa  xmm0, XMMWORD PTR .LC2[rip]
lea     rbx, [rsp+32]
lea     rbp, [rsp+48]
movaps  XMMWORD PTR [rsp+16], xmm0
paddb   xmm0, xmm1
movaps  XMMWORD PTR [rsp], xmm1
movaps  XMMWORD PTR [rsp+32], xmm0
```

**(b)** SIMD-ised instruction set

**Figure 2.6:** Explicit SIMD code and the instructions generated by the compiler.

data types, from 8-bit integer to 64-bit float. Nowadays, according to the Steam hardware survey, 100% of the CPUs observed support SSE2 (46) [1].

Intel Advanced Vector Extensions (AVX) was introduced in 2008 and first supported by the Sandy Bridge processor (47). It introduced 256-bit wide YMM registers, with the lowest 128 bits being identical to the XMM registers (48, Chapter 13). In 2013 AVX-512 was introduced with ZMM registers, which are 512 bits wide. From the Steam hardware survey, it can be seen that AVX and AVX2 are supported by roughly 90% of all CPUs observed. However, AVX-512 is supported by less than 10%.

SIMD instructions can be obtained in two ways, implicitly and explicitly. Implicit means that the compiler generates SIMD instructions based on code patterns, an example of this can be seen in Figure 2.5a. However, if the complexity of the code increases, it could be that the compiler cannot generate SIMD-ised instructions. Therefore, the alternative is to write explicit SIMD code, which is guaranteed to be converted to SIMD instructions by the compiler. The Intel® Intrinsics Guide (49) can be used as a reference to figure out which function should be used and the instructions that are generated with it.

An example of explicit SIMD code can be seen in Figure 2.6. As in Figure 2.5a, there are two 8-bit arrays $A$ and $B$ of length 16. Thus the total size is 128 bits for both arrays. The first two lines of Figure 2.6a are concerned with loading the arrays in the special XMM registers. `_mm_load_si128` corresponds with the `movedqa xmm` instructions seen in Figure 2.6b. `_mm_add_epi8` requires two vectors loaded into the XMM registers and adds packed 8-bit integers, storing the result in `C_vec` in this case. This line corresponds with the `paddb xmm` instruction in Figure 2.6b.

To make optimal use of SIMD instructions, knowing which specific instructions can be used is essential. To make the most out of SIMD instructions, we want to use the newest (AVX or AVX-512) as these provide the greatest speed-up. However, we need to know the

[1] It should be noted that this survey may not be entirely representative of the average computer user.

| Event | Latency | Scaled |
|---|---|---|
| 3GHz CPU cycle | 0.3 ns | 1 s |
| L1 cache access | 0.5 ns | 2 s |
| L2 cache access | 2.8 ns | 9 s |
| L3 cache access | 12.9 ns | 43 s |
| Main memory access | 120 ns | 6 min |

**Table 2.2:** Event with corresponding latency, also scaled to 1 second

specific CPU on which we execute the code to use the correct instruction set. Information such as the available SIMD instructions can be retrieved, allowing us to tailor the code to the CPU. One option is to generate different instructions at compile time. Another option is to use the open-source C library `cpu_features` (50), which allows us to retrieve CPU features at run-time.

## 2.5 Cache latency

The CPU cache can store small chunks of data that can be read orders of magnitude faster than from the main memory (51). The trade-off is that the cache is typically much smaller than the main memory. In a modern CPU, the cache consists of three levels, L1, L2, and L3. The higher the level, the slower the speed, but the more data can be stored. Typically, L1 and L2 are much smaller compared to L3 but, therefore, also much faster. L1 is split into two segments—L1d, which can be used for storing data, and L1i, which can be used for storing instructions.

When the CPU requests data, it first looks at whether this can be found in the L1 cache. If this is not the case, it looks in the L2 and L3 caches. When the data is found in one of these caches, it is referred to as a cache hit. When the data is not found in any caches, it must be fetched from the main memory, causing a cache miss. Fetching data from main memory is orders of magnitude slower than fetching data from any of the cache levels, as can be observed in Table 2.2.

# 3

# Related Work

In this chapter we first describe strategies to translate one query language to another. Next we highlight a survey that has been conducted on graph database management systems. Finally, we describe four papers who each describe database systems tailored to handling graph-like workloads.

## 3.1 Mapping Strategies

### 3.1.1 Mapping from Graph to Relational Queries

Research has been conducted on translating a given query language to another, such as translating graph query languages to SQL. An example of this is *Cytosm* (Cypher to SQL Mapping) by Steer et al. (37). It acts as an application to execute graph queries on RDBMSs. In addition, they introduced *gTop*. This format can capture the structure of property graphs and allow a mapping between a property graph and relational tables. Steer et al. find that the translated SQL queries executed on the Vertica columnar RDBMS show comparable performance to SQLGraph and vanilla Neo4j v2.3.4, depending on the type of graph query on a columnar RDBMS (37). SQLGraph allows Gremlin queries to be converted into SQL (27). It uses a combination of relational and non-relational data storage. The *adjacency information* uses relational storage, while the attributes regarding all the nodes and edges are stored with JSON storage.

GraphGen (52) 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. GraphGenQL specifies the queries, loosely based on SPARQL, Cypher, and PGQL (52).

Zhao and Yu showed that it was possible to support a large class of graph algorithms, such as BFS, Bellman-Ford, and PageRank in SQL (53). They introduce new semiring-based graph operators alongside the existing ones (selection, projection, union, set difference, Cartesian product, and rename) to provide explicit support for graph algorithms.

Lastly, there is IBM Db2 Graph by Tian et al. (54), which is an in-DBMS graph query approach that provides Gremlin support as an extension to Db2. Similar to GraphGen, it has a *graph overlay* method that exposes *graph views* of the relational data.

### 3.1.2 Mapping from RDF to PG

Thakkar et al. (55) identify a lack of essential interoperability between two query languages based on the two data models most commonly used for knowledge graphs. The query languages are SPARQL (56) for RDF data models and Gremlin, a property graph traversal language. The paper presents Gremlinator, which acts as a translator between SPARQL and Gremlin. The goal is to increase the interoperability between the expressive data modelling that is RDF and efficient graph traversals that are possible with property graphs. It claims the following advantages:

1. Existing SPARQL-based applications can switch to property graphs in a non-intrusive way.

2. It provides the foundation for mixed-use of RDF triple stores and property graph stores.

Gremlinator supports pattern matching and graph traversal queries and claims to be more general than Cypher since it provides a common execution platform that supports any graph computing system for addressing the querying interoperability issue (55). A limitation of Gremlinator is the lack of support for variables for property predicates.

Thakkar et al. provide an in-depth explanation on the inner workings of `sparql-gremlin` in (57). An implementation is available as a plugin for the Apache TinkerPop graph computing framework. The translation takes a SPARQL graph pattern as input and returns a Gremlin expression. For a given triple pattern ($v1$, $v2$, $v3$), the transformation to Gremlin is different, depending on whether $v2$ refers to a property or a relationship. `AND` graph patterns indicate a join between two tables, which is implemented in Gremlin as a `match` operator. A `FILTER` can be implemented in Gremlin as a `.where` clause. The

`SELECT` clause is implemented using the `.select` operator. Both Gremlin and SPARQL support several types of aggregates. During the experiments, the authors evaluated both the query correctness as well as the performance. They observed that in all 60 SPARQL queries tested, the corresponding Gremlin traversals provided identical results. In most cases, the performance of the Gremlin traversals showed to be competitive compared to the SPARQL query. However, Gremlin outperformed the SPARQL queries by two orders of magnitude for path queries and queries containing a *star*-shaped execution plan.

## 3.2 Survey on Graph Database Management Systems

Since the field of graph processing workloads is quite scattered (2, 58), many different GDBMSs exist, each having its strengths and weaknesses. Besta et al. (6) provide a survey and taxonomy of 45 graph database systems. The authors start by explaining the various graph models used by the surveyed systems; see Section 2.1 for a similar explanation.

The graph database systems were categorised into several categories. Firstly, their general backend type significantly impacts the other aspects of the graph database and is easily defined. Backend types considered are tuple stores, document stores, key-value stores, wide-column stores, RDBMS, or Object-Oriented DBMS. Further consideration was the conceptual graph data models and representations supported, data organisation, data distribution, query execution, transaction types, and the supported query languages.

Some systems use a specific backend (e.g. relational) technology to store the graph data. They then have an added frontend to query the graph data. Other systems are specifically designed to store graph data, also called native graph databases. These are based on either the LPG or RDF. A point worth mentioning is that almost all systems are closed-source or do not provide all the details on the internals of their system, making in-depth comparisons more difficult. The survey includes a comparison of features and languages supported by the systems discussed. The comparison regarding the supported query languages shows how fragmented the field is. There are six query languages, SPARQL, Gremlin, Cypher, SQL, GraphQL, and API (formulating queries using a native programming language such as C++). While some of these languages are similar, most systems only support one. Some systems do not even support any of the six mentioned here but support some different language(s).

The authors highlight some challenges related to GDBMSs. The first one is that there is no single graph model for the systems. The most widely used is LPG. However, it is not always fully supported. Secondly, the different graph database workloads' good design

choices are not yet determined. Some graph workloads, such as graph pattern matching problems or vertex reordering problems have been unaddressed by graph database system designers, hurting these systems' performance.

## 3.3 Data processing systems supporting graph processing workloads

This section highlights several data processing systems capable of handling graph workloads. We show their purpose and highlight some of their strengths and the challenges these systems have.

### 3.3.1 GRADOOP

Junghanns et al. present *GRADOOP* (59), a framework that extends Apache Flink and has the advantages of distributed graph processing. They identify two key categories of systems that focus on the management and analysis of graph data: graph database systems and distributed graph processing systems. Graph database systems focus mainly on efficient storage and transactional processing using a provided declarative graph query language. The problem with these systems is that they are unsuitable for high-volume data analysis and graph mining (59). On the other hand, parallel graph processing systems such as Google Pregel (60) can process large-scale graph data. However, these systems lack an expressive graph data model and declarative graph operations.

To combine the strength of both, GRADOOP was built. It uses the *Extended Property Graph Model*, which does not enforce any schema. *Extended* refers to the fact that nodes and edges may exist simultaneously in one or more graphs. The analytical programs are defined in the *Graph Analytical Language* (GRALA), a domain-specific language for the Extended Property Graph Model, and can be accessed using a Java API. It includes composable graph operators and general operators for data transformation and aggregation. The operators take graphs as input and also produce graphs, making them composable. Several well-known operators have been implemented, such as PageRank and connected components. GRADOOP uses a distributed dataflow system to achieve horizontal scalability; an example of such a system is Apache Spark (61). A brief demonstration of the system is provided. However, no actual results are presented, making it difficult to estimate the performance of GRADOOP.

### 3.3.2 The case against specialised graph analytics engines

Fan et al. (3) provide an answer to whether RDBMSs should be used for graph analytics instead of creating specialised graph engines. The authors present Graph Analysis in Legacy (Grail) as a syntactic mapping layer for any queries. It argues that the advantages of extending an RDBMS for "non-core relational" processing come with the benefits of building on top of mature technology that has been researched for decades and has proven robust. In addition, it is likely that enterprises already make use of an RDBMS in their ecosystem. Adding a system (a graph engine) would lead to more significant overhead and reduce overall work efficiency. The authors use a single-node Microsoft SQL Server 2014 that provides T-SQL (3). The users can interact with Grail through an API since writing graph analytical queries in SQL is not the natural way to do it. The query is mapped to a runnable T-SQL query using the translator, which is then optimised using the optimiser. The data is stored similarly to SQL/PGQ; the edges and vertices are stored as tables. The computational model is a vertex-centric approach and uses intermediate tables, in which data is stored that is required for every iteration of a computation.

Apache Giraph originated from Google Pregel (60) and GraphLab (62), which is mainly designed for parallel machine learning algorithms, are used to compare the performance of Grail. Three queries are conducted, one related to the single-source shortest path, one related to PageRank, and one related to Weakly Connected Components. The datasets contain 9k to 41 million vertices and between 5 million and 1.46 billion edges. The authors show that Grail is slower than GraphLab on the smaller datasets, though it scales better for larger datasets. The figure presented shows both the computation and loading time. It appears that GraphLab has a relatively high loading time, whereas SQL Server has a high computation time. It is unclear what is precisely meant by loading time. If it refers to loading the datasets, it should have been considered that this time is amortised if the algorithms are conducted multiple times. In that case, GraphLab would show equal performance compared to SQL Server. It is also unclear how well optimised the queries for GraphLab and Giraph were. Therefore, the results may not be suitable for system-to-system comparisons.

### 3.3.3 GRainDB

Jin et al. present GRainDB (12), a system that extends the RDBMS DuckDB by providing graph modelling, querying, and visualisation capabilities. The authors modified the internals of DuckDB to integrate storage and query processing techniques to make

the workload on graphs more efficient. These modifications include predefined pointer-based joins, hybrid graph-relational data modelling and querying, and graph visualisation. Performing in-depth modifications on DuckDB appears to be a risky strategy, as this is still a relatively new database system in which the internals are still prone to change quite frequently. By making modifications, future versions of DuckDB become incompatible with the modified version, and updating becomes difficult.

The team identifies advantages for relations and graphs in the data model. Relations can represent n-ary relationships for arbitrary values of n, providing a natural structure to model normalised data. However, the authors argue that the most popular query language, SQL, is considered very cumbersome for graph-based queries. Most notably, recursive queries are challenging to write and understand.

Since graph queries often involve joins, the authors looked at improving the join capabilities of the RDBMS. Three techniques were implemented: Predefined pointer-based joins, factorisation, and worst-case optimal join algorithms. Furthermore, the authors extended SQL to support nodes, edges, and path patterns using inspiration from Cypher and GSQL.

### 3.3.4 MillenniumDB

Vrgoč et al. present MillenniumDB, a persistent, open-source, graph database (63). It is based on domain graphs, which can act as an abstraction on which other popular graph models, such as RDF or the LPG can be supported. The main strength of the domain graph model is that it better captures higher-arity relations more directly; more details of the model are provided in (63). The engine is based on relational database management systems techniques, worst-case-optimal join (64) and graph-specific algorithms.

The authors set several goals for MillenniumDB. First, it should be able to support various graph database models and query languages by generalising them. Second, it should make use of state-of-the-art techniques. Third, MillenniumDB should be easily extensible. Therefore a modular design is used. Finally, it should be open-source, such that other researchers can reuse the techniques. MillenniumDB aims to support multiple graph query languages. However, the authors found that none of the existing ones can fully utilise the properties of the domain graph model. Therefore, the authors created their graph query language, DGQL, which closely resembles Cypher and contains SPARQL features.

To benchmark the performance of MillenniumDB, the authors used the WikiData knowledge graph (65). They compared their performance with three RDF engines: Jena TDB (66), BlazeGraph (67), and Virtuoso (68), as well as the property graph database

Neo4j Community Edition (69). It finds that the execution times of graph pattern matching queries are consistently faster for MillenniumDB compared to the other database engines.

# 4

# Design & Implementation

In this chapter we will describe key components related to implement efficient path-finding operators in DuckDB. Figure 4.1 shows a schematic overview of the components. Step ① shows the `CREATE PROPERTY GRAPH` which defines the property graphs over the relational tables previously created with SQL. Step ② is the path-finding query which leads to the creation of a Compressed Sparse Row (CSR) data structure through steps ③–⑤, and to the final query results through ⑥ and ⑦.

The steps involving ④–⑥ have been designed and implemented by Singh et al., and are discussed in more detail in Section 4.2. Step ③ relabels the vertex ids into a contiguous range of ids and offsets to represent the graph with good memory locality and neighbourhood lookups. Step ④ builds a CSR data structure, see Section 4.4, for an unweighted graph ⑤. Step ⑥ then uses the CSR to compute the reachability between vertices using Multi-Source Breadth-first Search (MS-BFS).

This thesis extends the previous work in multiple ways. First, support is added to compute the shortest path length using the MS-BFS algorithm, see Section 4.5, as well as returning a list of vertices found on the shortest path in the *any shortest path* function, see Section 4.6. Second, support is added for building a CSR for weighted graphs. Using the weighted CSR, the weighted shortest path (cheapest path) can be computed using the batched Bellman-Ford algorithm discussed in Section 4.7. The output for the path-finding operations is using in step ⑦.

Step ③ and step ⑦ will make use of an optimisation introduced in Section 4.3 which allow for the hash table used to perform join operations to be shared between multiple joins. In both step ③ and step ⑦ we perform two join operations that create the same hash table. Instead of building the same hash table twice, we will build one, and reuse it for the second join operation.

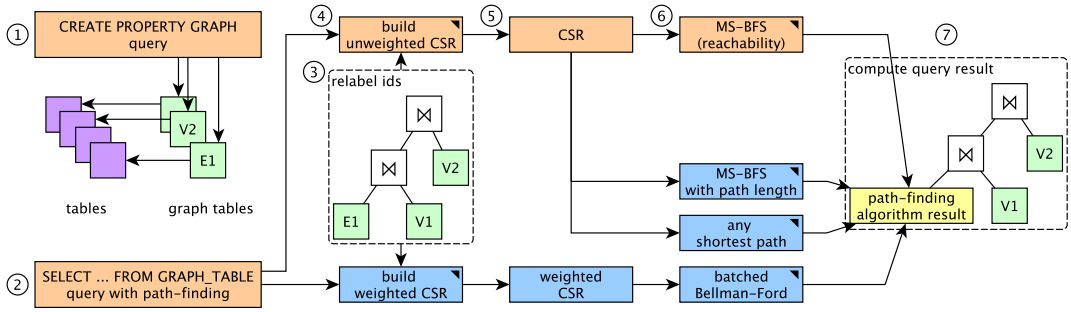



**Figure 4.1:** Schematic overview of key components for efficient path-finding operators

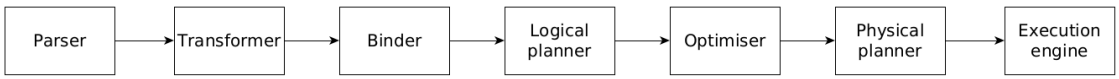



**Figure 4.2:** Execution pipeline of DuckDB

## 4.1 DuckDB

DuckDB's architecture is a database management system specialised in analytical workloads, performing queries with joins, aggregations, and filtering operations on up to 100s of GB of data (20). Like SQLite, DuckDB is an in-process system, though SQLite is specialised in transactional workloads.

DuckDB consists of several components: parser, transformer, logical planner, optimiser, physical planner, and execution engine, see Figure 4.2. The system can be accessed through a C/C++ API and a SQLite compatibility layer.

### 4.1.1 DuckDB execution pipeline

The SQL parser is based on the PostgreSQL SQL parser (20). 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.

The optimiser will then create an optimised logical plan which is given to the physical planner, turning it 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 (70). A join operator that uses 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. For example, a pipeline with a dependency is one containing a *join* operator. The start of a pipeline is referred to as a source. The end is referred to as the sink, where all the data is collected (materialised). Only sink operators, such as sorting, building a hash table, and aggregation, need to access all the data before proceeding. All other operators do not need to materialise 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-materialising operation, it can be scheduled in the middle of a pipeline.

The execution engine of DuckDB is vectorised (20). The use of vectors is more CPU efficient than the more common tuple-at-a-time execution found in other DBMSs as it amortises the interpretation cost (71). A vector typically contains 1024 elements. Thus if a function contains less than or equal to 1024 elements, these can be placed in a single vector. Whenever a function contains more than 1024 elements, another vector is created for every 1024 elements. The size of vectors can be adjusted internally, though this is typically only done for debugging purposes.

### 4.1.2 Join operator

For every join operator, a hash table needs to be built on which the join can be performed[1]. The operator needs to wait for the entire hash table to be built. Only then can the join be correctly executed. Similarly, another pipeline might require the outcome of this join before it can be executed, creating a chain of dependencies. Cardinality estimation is performed to assess which of the two tables is the smaller one. These are estimations; thus, it cannot be guaranteed that the smallest table is always used.

This smallest table is then used to build a hash table, 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, 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.

[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. Using an array instead of a hash table is referred to as a perfect join (72).

### 4.1.3 Design decisions

In the current implementation, it is only possible to create scalar User-Defined Functions (UDFs) in the extension modules. These scalar UDFs are as fast as the built-in functions of DuckDB due to the vectorised query processing, whereby DuckDB handles the parallelisation of the UDF. We will make use of scalar UDFs for the path-finding and CSR creation functions to create a lightweight implementation.

The parallelisation for scalar UDFs is done using the morsel-driven method (73). DuckDB will spawn a thread for roughly every 100 000 rows, depending on the size of the standard vector. This threshold can be adjusted using the pragma `verify_parallelism`, though this is only used for debugging purposes. With a pragma, the internal properties of the database engine can be adjusted.

For each table, the `rowid` is a pseudo column that stores the row identifier based on the physical storage. These ids are dense integers starting from zero up to the number of rows in the table. If rows are deleted, they create a gap in the rowid which may be reclaimed later.

In DuckDB, it is possible to store client-related information in the `client context`. For example, this information can be related to pragmas that have been enabled or disabled, such as profiling options. The data inside the client context is stored for an entire session and can thus store information spanning multiple queries.

### 4.1.4 Testing in DuckDB

In DuckDB, unit tests can be written using the *SQLLogicTests framework* to test the correctness of the implemented functions (74). This test suite is not intended for benchmarking the performance of SQL queries.

## 4.2 Current state of SQL/PGQ in DuckDB

This thesis will use the work done by Singh et al. at CWI (22). They identified several challenges that needed to be addressed. DuckDB is primarily intended for tabular workloads and its developers want 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 the correct parsing of SQL/PGQ queries. One of the first challenges was successfully parsing SQL/PGQ queries. Modifications were made to the DuckDB parser to allow for the ASCII-art style query syntax introduced with SQL/PGQ as shown in Listing 2.7.

New keywords like `GRAPH`, `LABEL` and `PROPERTIES` were added to the parser to allow correct parsing of SQL/PGQ queries. The SQL/PGQ queries are transformed into SQL queries, an example of which can be seen in Listing A.1 and Listing A.2.

Another consequence of limiting the core features was the decision to implement the new operators, such as the batched MS-BFS algorithm (75) described in Section 4.5, as scalar UDFs. A benefit of implementing these operators as scalar UDFs instead of expressions is that little to no changes have to be made to the internals of DuckDB. No further changes needed to be made to the parser to implement reachability, and no new logical and physical operators had to be introduced.

## 4.3 Shared hash join

Queries can contain multiple expensive joins which are often identical. Identical joins can also occur in graph-like queries, where it is common to join the vertex table twice on the edge table: once for the edge's source and once for the its destination, see Section 2.1.2.1.

In the current state of DuckDB, a hash table is built from the smaller table for each join. However, this is wasteful when 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 Compressed Sparse Row (CSR) data structure representation for the edge table, see Section 4.4. Thus, it is necessary to perform two joins, namely `person1_id` $\bowtie$ `person_id` and `person2_id` $\bowtie$ `person_id`. In this case, the right side is identical in both joins. See Figure 4.3 for the relevant part of the physical plan of this example query. An optimisation is to build this hash table only once and reuse it for any identical joins containing the same sink. This optimisation will eliminate the need to build the same hash table multiple times.

During the building phase of the pipelines, we will save all join operations we come across in combination with their respective pipeline. If a duplicate join is encountered, the sink state of that pipeline will 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 optimisation 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.

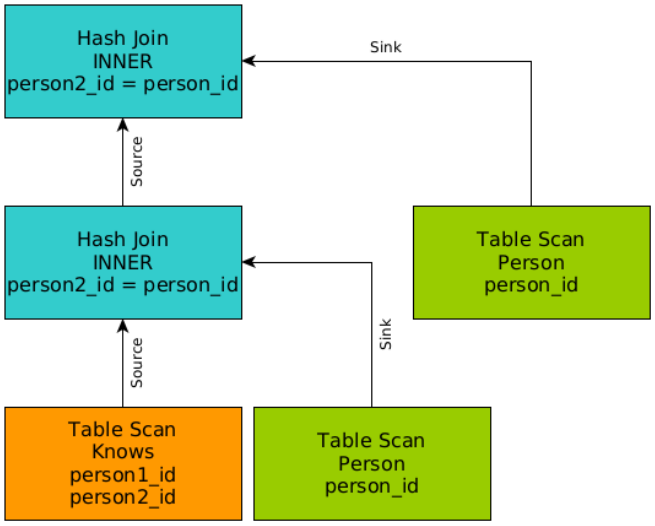



**Figure 4.3:** Example of a duplicate sink state in the physical plan

The initial idea was to detect the duplicate joins in the optimiser. The optimisation can be done by adding a rule to the optimiser, which scans for duplicate joins. In the optimiser, 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. If 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 `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 several challenges with this approach. First, 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 use different columns. These would then result in non-duplicate hash tables, which can not be reused by each other. Thus it is essential to also look at the conditions of the joins.

Assuming the duplicate join can be detected, a 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 pipeline. This would have created additional work that could be saved using another approach ultimately chosen as the best-suited option.

Instead of detecting the duplicate joins in the optimiser, they are detected during the pipeline creation. At this point, the physical plan has been created and converted into pipelines. When the physical operator `HASH JOIN` is detected, we save the corresponding

pipeline, and the `HASH JOIN` operator is 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 a duplicate of those seen earlier. This check 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.

## 4.4 Compressed Sparse Row

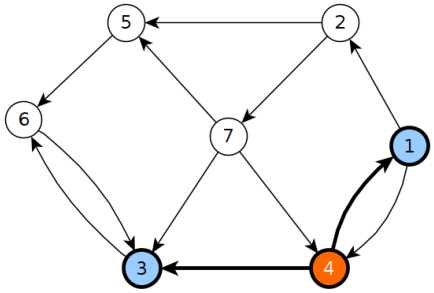
The Compressed Sparse Row (CSR) data structure can be used to create a compact data structure with a good locality. The locality is vital for the shortest and cheapest path algorithms, as these algorithms are not computationally heavy, but they require an excellent memory locality. Therefore, it is essential to reduce the number of cache misses.

An alternative is a pointer-based data structure. In this case, every node contains a mini-index to all nearby nodes (76). However, this becomes inefficient in the case of multiple traversals due to the random memory access leading to poor memory locality.

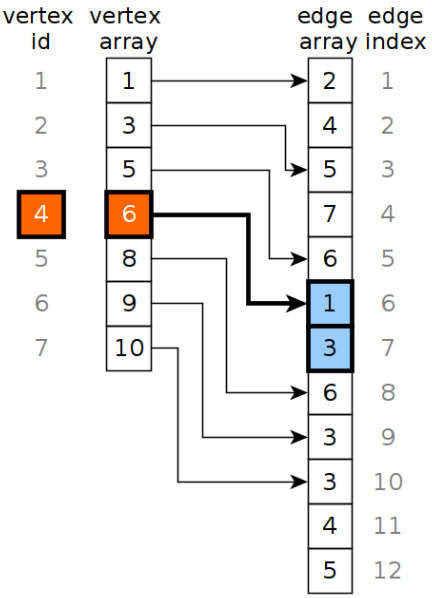
The memory locality can be improved by using the CSR data structure. In this case, we use the row identifiers of the vertex table. The ids of the vertices in the vertex table need not be in order for the CSR to work. The fact that no sorting is required is important for parallelisation since queries containing an `ORDER BY` (a sort) can typically not be executed in parallel.

Figure 4.4 shows an example of the CSR data structure for an unweighted directed graph. The CSR consists of the vertex array and the edge array. In the vertex array, at the index of a vertex id, the offset in the edge array at which the vertex id of the outgoing edges corresponding to this vertex starts is stored. Consider the highlighted vertex 4, which has outgoing edges to vertices 1 and 3. In the vertex array at index 4, we store the value 6. This value can be used as a lookup value in the edge array. At index 6 in the edge array, we find the value 1, corresponding to the first outgoing edge of vertex 4, namely vertex 1. The next value in the edge array is 3, corresponding to the second outgoing edge of vertex 4, namely vertex 3. We know that vertex 4 has two outgoing edges because the offset in the vertex array for vertex 5 is 8. Thus we stop looking for outgoing edges once the index in the edge array reaches 8.

The CSR will be created on-the-fly just before the shortest or cheapest path function is executed. The alternative is creating the CSR whenever a vertex/edge table is created or updated. However, this was not chosen for two reasons. First, CSR creation will likely not take much time relative to the shortest and cheapest path functions. Second, it is difficult




**(a)** Unweighted directed graph




**(b)** CSR representation of unweighted directed graph

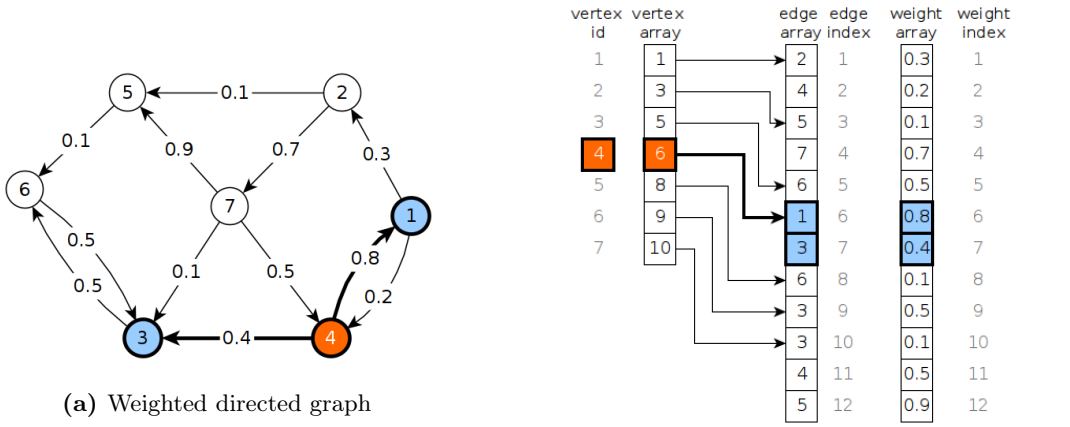
**Figure 4.4:** Unweighted directed graph and the CSR representation

to update the CSR whenever vertices or edges are deleted or added to their respective tables. Since updating is difficult, it is more likely that the entire CSR structure will be recreated from scratch every time an update is made to the vertex or edge tables.

In DuckDB, the CSR is created in two separate scalar UDFs to avoid `ORDER BY` and thus allow the creation to be done in parallel. Upon creation, a CSR is stored in the client context. It is possible to create multiple CSR structures and store them all in the client context. After the shortest or cheapest path functions have been finalised, the CSR will be deleted. This is currently done using a pragma `delete_csr_by_id`. Each CSR gets assigned an ID, so the shortest and cheapest path functions know whether the CSR has been initialised and where to find the CSR.

The first UDF is concerned with initialising the vertex array. The vertex array consists of atomic 64-bit integers. It requires an ID, the number of vertices in total and for every vertex, the row id and the number of outgoing edges. At first, the vertex array will only hold the number of outgoing edges per vertex, which will be converted to offsets for the edge array later. Returned will be the number of outgoing edges per vertex.

The second UDF is related to initialising the edge array, which can only be done after the vertex array has been initialised. It sets all the offsets in the vertex array and fills the edge array with the destination vertices. It requires the CSR ID, the number of vertices, the




**(a)** Weighted directed graph

**(b)** CSR representation of weighted directed graph

**Figure 4.5:** Weighted directed graph and the CSR representation

number of edges for every vertex and the vertex table row ids of all source and destination vertices found in the edge table.

Extending the CSR structure also to support edge weights can be done by adding a third array. This third array will store the edge weights corresponding with the edges in the edge array. The current implementation in DuckDB supports 64-bit integers and doubles to be stored as edge weights. Storing the weights in the CSR is done with the same UDF as the edge array. In DuckDB, it is possible to overload UDFs, with the function to initialise the edges taking an additional parameter, namely the weights of the edges.

## 4.5 Shortest Path

Computing the shortest path between two vertices in an unweighted graph can be done using Multi-Source Breadth-first Search (75) (MS-BFS), pseudo-code is provided in Algorithm 5. This is an extension of the textbook BFS discussed in Section 2.3. Using MS-BFS, multiple BFS instances are exploring the graph together. Whereas textbook BFS is a single-source algorithm, MS-BFS is a multi-source algorithm. This algorithm is useful as multi-source shortest path queries will be common in SQL/PGQ. Executing these queries one by one would be inefficient. Moreover, with MS-BFS, we can use the vectorised execution engine of DuckDB, possibly also parallelising the execution of MS-BFS. Since the MS-BFS algorithm does not involve heavy computations, a major bottleneck will likely be memory access. This bottleneck is why the CSR data structure was created for the MS-BFS algorithm.

**Algorithm 5** Multi-Source Breadth-first Search

**Input:** $G, \mathbb{B}, s$
$seen_{s_i} \leftarrow \{b_i\}\ for\ all\ b_i \in \mathbb{B}$
$visit \leftarrow \bigcup_{b_i \in \mathbb{B}} \{(s_i, \{b_i\})\}$
$visitNext \leftarrow \varnothing$
**while** $visit \neq \varnothing$ **do**
    **for each** $v$ **in** $visit$ **do**
        $\mathbb{B}'_v \leftarrow \varnothing$
        **for each** $(v', \mathbb{B}') \in visit$ **where** $v' = v$ **do**
            $\mathbb{B}'_v \leftarrow \mathbb{B}'_v \cup \mathbb{B}'$
        **end for**
        **for each** $n \in neighbours_v$ **do**
            $\mathbb{D} \leftarrow \mathbb{B}'_v \setminus seen_n$
            **if** $\mathbb{D} \neq \varnothing$ **then**
                $visitNext \leftarrow visitNext \cup \{(n, \mathbb{D})\}$
                $seen_n \leftarrow seen_n \cup \mathbb{D}$
                Do BFS computation on $n$
            **end if**
        **end for**
    **end for**
    $visit \leftarrow visitNext$
    $visitNext \leftarrow \varnothing$
**end while**

In the implementation made by Singh et al. it was possible to compute the reachability of a node using the MS-BFS algorithm. The reachability implementation required a bitmap for every source. The bitmap has a length equal to the number of vertices in the graph. The bit relating to a vertex would be set to 1 if it was reachable from a given source. Due to the bitmap and writing implicit SIMD code, it is possible to execute 128, 256, or 512 BFS instances using only a handful of instructions. Depending on the CPU on which this is executed and the compiler options given, 128 instances would be done in the case of SSE/SSE2, 256 in the case of AVX, and 512 in the case of AVX-512.

When we wish to compute the shortest path, we need to allocate more than a single bit for every vertex, as the distance is very likely to be greater than 1. However, keeping in mind that we wish to write SIMD-friendly code, we should keep the space allocated for the vertices as small as possible. For example, if we allocate 8 bits for every vertex and use SSE/SSE2 instructions, it is only possible to execute $128/8 = 16$ BFS instances simultaneously.

We know that the shortest distance can never be a negative number. Therefore we can use unsigned integers, doubling the maximum length we can store with the same amount of bits. If we use 8-bit unsigned integers, the maximum length of a path can be 255. Though we need to reserve one number to signal that a vertex is unreachable, thus the maximum length will be 254.

Another reason to use fewer bits to keep track of the distance is that graphs often have the small-world property, meaning that the distance between any two vertices is small compared to the size of the graph (77). Therefore, it is not necessary to use 64-bit integers as it is not likely that the path length will be that high. Additionally, each BFS discovers most vertices in a few iterations (75), and concurrent BFSs have a high chance of overlapping sets of discovered edges in the same iteration. This overlap allows memory access to be shared among the multiple BFSs and reduces the chance of cache misses, reducing the overall computation time (75).

In the current implementation, we start the algorithm with 8-bit unsigned integers to keep track of the distance. It could be argued that 4-bit unsigned integers would also suffice as a starting point. However, there exist no SIMD instructions for 4-bit integers (49). The arrays used to hold the path lengths for the vertices are initialised with the maximum 8-bit unsigned integer value, 255, to indicate that these vertices have not been reached yet. If the path length is longer than 254, we copy the 8-bit unsigned array to a 16-bit unsigned integer array to avoid integer overflow. This pattern is repeated to 32-bit if we find that 16-bit unsigned integers are also not enough to store the path length. However, every

time we copy the array to larger arrays, we reduce the efficiency of SIMD instructions. As mentioned, with 8-bit integers, we can run $128/8 = 16$ BFS instances at once. With 32-bit integers we are only able to run $128/32 = 4$ BFS instances.

Figure 4.6 shows an example. Two BFS instances will be run, one starting from vertex 1, marked in green, and the other starting from vertex 7, marked in blue. The algorithm comprises three arrays for every source. The *visit* array is a bitmap that contains the currently active vertices for which neighbours will be explored for the next iteration. The *seen* array, also a bitmap, holds information on all vertices that have been seen so far. The *depth* array tells us in which iteration the vertex was discovered, i.e. the path length.

For B1 in Figure 4.6, we set the bit at position 1 in both the visit and seen arrays. Additionally, we set the depth from position 1 to zero. The same is done for B2, where the bits at position 7 are set to zero.

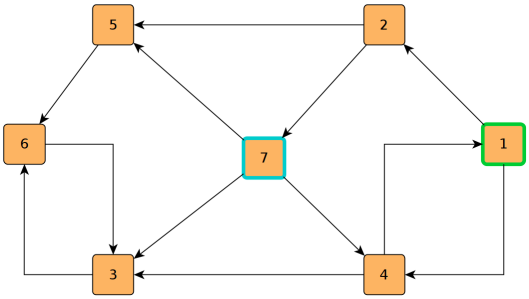
For the next step in B1 and B2, we explore the neighbours for the active vertices in the visit array. In step 1, shown in Figure 4.7, for B1, we explore the neighbours of vertex 1, namely 2 and 4. We set the bits in visit and seen for 2 and 4 respectively and unset the bit for 1, seen in Figure 4.7b. We mark that the distance for these is one. For B2, we do the same for the neighbours of vertex 7, which are 3, 4, and 5.

In step 2, shown in Figure 4.8 we see that vertex 4 is active for both B1 and B2. The neighbours of vertex 4 are 1 and 3. For B1, vertex 1 has already been discovered (as this was the source), which guarantees that no shorter path can be found. We, therefore, do not update the distance and do not need to explore this vertex for this BFS instance. For B2, we have already discovered vertex 3, which had a length of 1 from the source vertex 7. We, therefore, also do not explore vertex 3 further for B2.

In step 3, shown in Figure 4.9, we discover the last vertices for both B1 and B2. At that point, we observe that no more edges lead to unseen vertices from these discovered vertices, and thus we terminate the algorithm.

## 4.6 Any Shortest Path

With *any shortest path* we wish to find the shortest path and return all the nodes on this shortest path. The input to this function is the CSR, the row ids of the source and destination nodes. Returned will be a list containing the row ids of the nodes, including the source and destination, found on the shortest path. If there are multiple shortest paths of equal length, we will only return a single one. This algorithm will also use the MS-BFS described in Section 4.5. However, the MS-BFS algorithm needs to be extended to keep



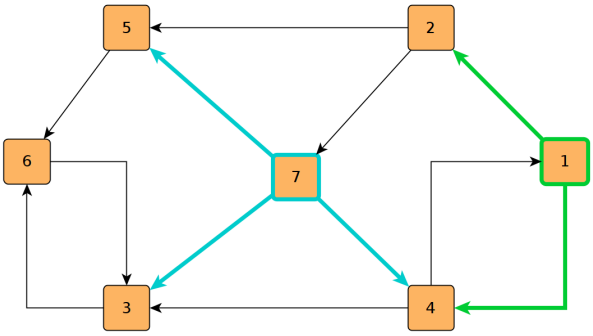

**(a)** Graph representation of initial state

Initial State

| Visit | | | Seen | | | Depth | |
|---|---|---|---|---|---|---|---|
| B1 | B2 | Index | B1 | B2 | Index | B1 | B2 |
| x | | 1 | x | | 1 | 0 | |
| | | 2 | | | 2 | | |
| | | 3 | | | 3 | | |
| | | 4 | | | 4 | | |
| | | 5 | | | 5 | | |
| | | 6 | | | 6 | | |
| | x | 7 | | x | 7 | | 0 |

**(b)** Array representation of initial state

**Figure 4.6:** Initial state



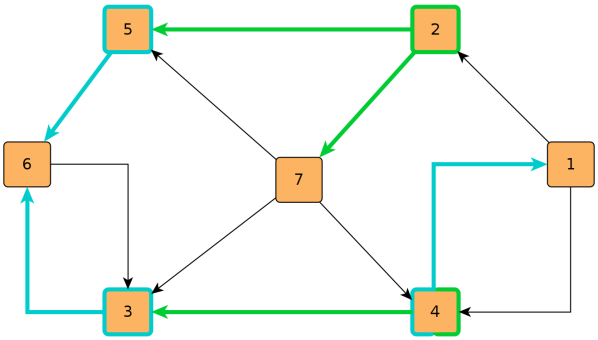

**(a)** Graph representation of step 1

1st level

| Visit | | | Seen | | | Depth | |
|---|---|---|---|---|---|---|---|
| B1 | B2 | Index | B1 | B2 | Index | B1 | B2 |
| | | 1 | x | | 1 | 0 | |
| x | | 2 | x | | 2 | 1 | |
| | x | 3 | | x | 3 | | 1 |
| x | x | 4 | x | x | 4 | 1 | 1 |
| | x | 5 | | x | 5 | | 1 |
| | | 6 | | | 6 | | |
| | | 7 | | x | 7 | | 0 |

**(b)** Array representation of step 1

**Figure 4.7:** MS-BFS step 1



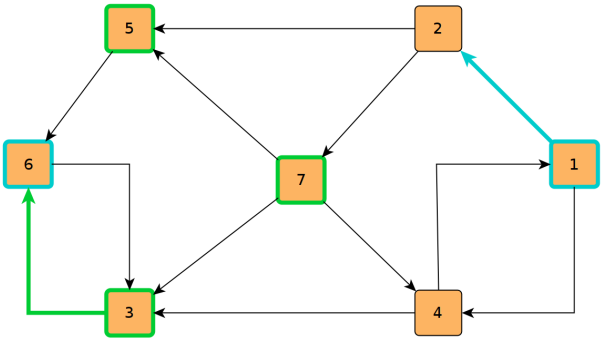

**(a)** Graph representation of step 2

2nd level

| Visit | | | Seen | | | Depth | |
|---|---|---|---|---|---|---|---|
| B1 | B2 | Index | B1 | B2 | Index | B1 | B2 |
| | x | 1 | x | x | 1 | 0 | 2 |
| | | 2 | x | | 2 | 1 | |
| x | | 3 | x | x | 3 | 2 | 1 |
| | | 4 | x | x | 4 | 1 | 1 |
| x | | 5 | x | x | 5 | 2 | 1 |
| | x | 6 | | x | 6 | | 2 |
| x | | 7 | x | x | 7 | 2 | 0 |

**(b)** Array representation of step 2

**Figure 4.8:** MS-BFS step 2




**(a)** Graph representation of step 3 (final)

3rd level

| Visit | | | Seen | | | Depth | |
|---|---|---|---|---|---|---|---|
| B1 | B2 | Index | B1 | B2 | Index | B1 | B2 |
| | | 1 | x | x | 1 | 0 | 2 |
| | x | 2 | x | x | 2 | 1 | 3 |
| | | 3 | x | x | 3 | 2 | 1 |
| | | 4 | x | x | 4 | 1 | 1 |
| | | 5 | x | x | 5 | 2 | 1 |
| x | | 6 | x | x | 6 | 3 | 2 |
| | | 7 | x | x | 7 | 2 | 0 |

**(b)** Array representation of step 3 (final)

**Figure 4.9:** MS-BFS step 3 (final)

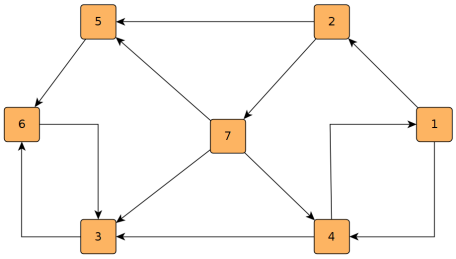
track of the nodes on the shortest path. For simplicity, the explanation will make use of a single-source BFS.

We start with an empty array $p$ of length $|V|$ where $|V|$ is the number of nodes in the graph. This array can reconstruct the final path once the algorithm terminates. Additionally, we keep track of the currently active nodes in an array $w$ of length $|V|$.

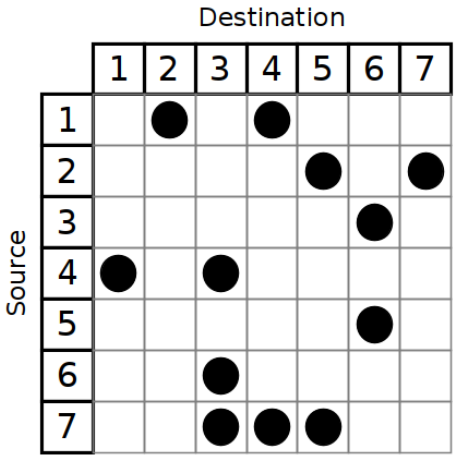
The first step is to denote the starting node in the array $p$. The starting node will be used when reconstructing the path once the algorithm has terminated. Denoting the starting node can be done by inserting a value that cannot be used anywhere else in the array, such as $-1$ since we know that row ids can never be negative. Thus, when reconstructing the path, if we reach $-1$, we know that we have found the original starting node. Once a value has been set in the array $p$, we need to ensure not to overwrite this value in the future. If we come across the same node at a later step, we always know that this is from a longer path, which is not of interest. Therefore, we set a boolean $is\_set$ to true to indicate that this value should not be changed later.

Every iteration, there are one or more nodes active. In $w$, we note the active node(s) by entering their index at their respective position in the array. For example, if node 1 is active, there will be a 1 at position 1 in the array $w$. The next step is to explore the neighbours of the active nodes. For every neighbour, we denote their parents' row id at the neighbour's index in the array $p$. The next step is to make the current neighbouring nodes active and update $w$. This step is similar to updating the $visit\_next$ and $visit$ arrays in the BFS algorithm.

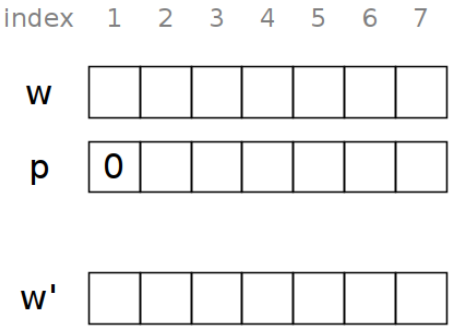
Figure 4.10 shows the initial state where no algorithm steps have been taken. The source from which the algorithm starts is node 1. In this algorithm, the destination node is only




**(a)** Graph representation of initial state



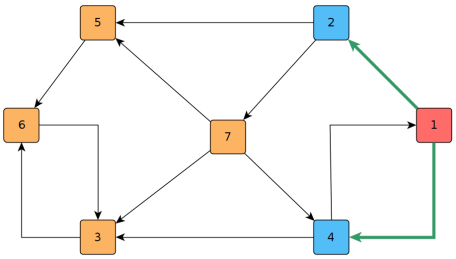

**(b)** Matrix representation of initial state



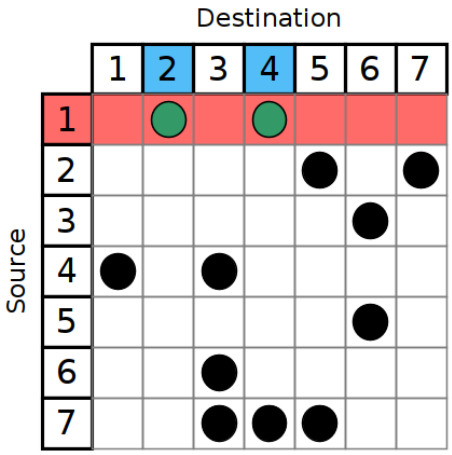

**(c)** Initial arrays

**Figure 4.10:** Any shortest path initial state

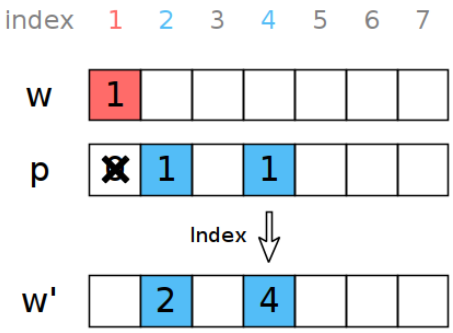
relevant after the entire graph has been explored and the path needs to be recreated, therefore the destination vertex is not included in the example.




**(a)** Graph representation of step 1




**(b)** Matrix representation of step 1

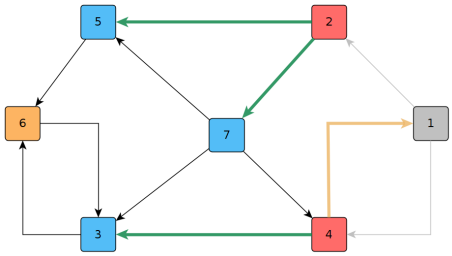



**(c)** Arrays after step 1

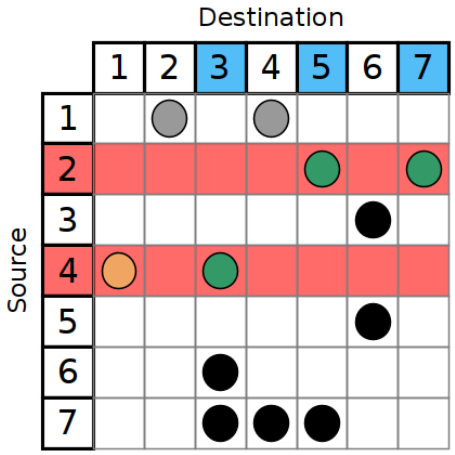
**Figure 4.11:** Any shortest path step 1

Figure 4.11 shows the first step, where node 1 is set as the active node in array $w$, seen in Figure 4.11c. Additionally, the neighbours of 1, nodes 2 and 4 are explored. In array $p$, it is noted that the parent of nodes 2 and 4 is 1. In array $w'$, we denote the indices of the nodes that have just been set in the array $p$, which in this case are 2 and 4.

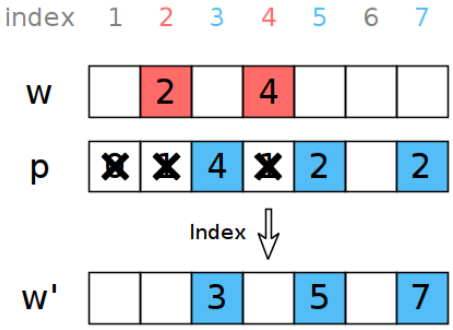
For the next step, shown in Figure 4.12, we observe that nodes 2 and 4 are now active in the array $w$, and nodes 3, 5, and 7 are being explored. The parent of node 3 is 4. Hence at index 3 in the array $p$, we denote 4 as the parent. For nodes 5 and 7 the parent is node 2. Hence at positions 5 and 7, the denoted value is 2. We also observe that one of the children of node 4, namely node 1, has already been explored in a previous iteration. We do not note that node 1 can be reached through node 4, since it would always be a longer path than what is currently known.




**(a)** Graph representation of step 2




**(b)** Matrix representation of step 2




**(c)** Arrays after step 2

**Figure 4.12:** Any shortest path step 2

For a complete run-down of the algorithm, see Appendix B.

In DuckDB, it is possible to return a column of type List (78). Therefore, every row returned will be a list containing all nodes in the path. Once the algorithm has terminated, it can reconstruct the path from a given destination to the source. Insert the id of the destination node in the output array $o$. Then in array $p$, take the value at the index of the destination node and add this to $o$. The next step is to take the value at the index of the previous value and add this to $o$. Repeat this until the index is equal to the id of the source node. Array $o$ then contains all values on the path between the source and destination nodes. In the example shown, node 1 is the source node and node 6 is the destination node. Thus, the array $o$ contains the value `[6]`. At index 6 in array $p$ we see the value 3, thus array $o$ now contains the values `[3, 6]`. Then, nodes 4 and 1 are added to $o$, resulting in the path `[1, 4, 3, 6]`.

The algorithm discussed so far is single-source. However, executing the algorithm sequentially for every source-destination pair would be inefficient. Therefore, we extend the multi-source algorithm, similar to the breadth-first search algorithm. This extension can be done by converting the arrays $w$, $p$, and $w'$ into dictionaries. Thus, we have the dictionaries $d_w$, $d_p$ and $d_{w'}$. For every entry in these dictionaries, the key corresponds to the id of the source node. The values are the arrays equal to the single-source version.

## 4.7 Cheapest Path

When every edge has assigned a weight, and thus we have a weighted graph, it is possible to compute the cheapest path from a source to a destination. As described in Chapter 2, multiple algorithms exist to compute the cheapest path, such as Dijkstra's algorithm and Bellman-Ford. Similarly to the shortest path, we wish to use a batched algorithm that allows us to evaluate multiple source-destination pairs at a time. A SQL/PGQ query would typically contain multiple source-destinations pairs. Using a single-source version of the algorithms mentioned above, we would have to execute the entire algorithm for every source. This sequential execution would create no opportunity for vectorising the code and create more inefficient memory access patterns.

Then et al. (79) have proposed two algorithms to compute the cheapest path, batched Bellman-Ford and Batched Dijkstra's algorithm. Implementing a batched version of Dijkstra's algorithm is more complex due to the Fibonacci heap required for every algorithm instance. Multiple instances will be run during the execution, each having a unique Fibonacci heap. These heaps' structure is different, limiting the possibilities of using

SIMD instructions. Then et al. also find that the performance of batched Bellman-Ford is 3–10× higher compared to batched Dijkstra's algorithm (79). Therefore, it was decided to implement the batched Bellman-Ford algorithm proposed by Then et al., as shown in Algorithm 6.

**Algorithm 6** Directed batched Bellman-Ford Algorithm

```
   Input: WeightedGraph G, Array<Vertex> sources
2: Output: VertexProperty<BatchVar<double> > dists
   VertexProperty<BatchVar<bool> > modified = false
4: dists = Infinite
   for i=1..sources.length do
6:     Node v = sources[i]
       dists[v][i] = 0
8:     modified[v][i] = true
   end for
10: bool changed = true
   while changed do
12:    changed = false
       for each v in G.vertices do
14:        if not modified[v].empty() then
               for each n in G.neighbours(v) do
16:                double weight = edgeWeight(v,n)
                   for each i in modified[v] do
18:                    double newDist = min(dists[n][i], dists[v][i] + weight)
                       if newDist != dists[n][i] then
20:                        dists[n][i] = newDist
                           modified[n][i] = true
22:                        changed = true
                       end if
24:                end for
               end for
26:        end if
       end for
28: end while
```

Similar to MS-BFS, multiple instances of Bellman-Ford will be executed simultaneously, allowing the memory access to be shared. For every source vertex, an array, referred to as *lane*, is created with a length equal to the number of vertices in the graph. This graph will

contain the distances from the source to all other vertices. During initialisation, the value at the index of the source vertex is set to zero. All other values are set to $\infty$ to indicate that they are unreachable. In the DuckDB implementation, we use the maximum possible value of the type we use, either a 64-bit integer or double.

We iterate over all the vertices in the graph, and for every vertex $v$, we evaluate the edges to all its neighbours $n$. Every $(v, n)$ edge has a weight $w$. Recall that we created a CSR structure containing a vertex array in order of the row identifiers. The values in the vertex array point to the starting location for their respective edges, which are also in order. Hence, we traverse the CSR vertex and edge arrays in order, providing a good memory locality. An issue arises when the distances of the neighbour (`dists[n]` in Algorithm 6) are retrieved. Accessing the neighbour is random access in the memory as we cannot guarantee that the neighbours are closely aligned. For an arbitrary vertex $v$, it could be that the first neighbour is close to $v$ in the `dists` array, while the second neighbour is not. The outgoing edges in the CSR edge array are in order, which improves the locality in the best case. However, no guarantee can be given and thus, most cache misses will occur when retrieving `dists[n]`.

Then for every lane $i$, we check if the currently cheapest known distance $dists[n][i]$ is higher than the distance known at $dists[v][i] + w$. If this is the case, we have found a new cheapest path and update $dists[n][i]$ accordingly.

A possible optimisation was introduced by Yen (43) where a bitmap is created for every lane, equal to the length of the number of vertices. A bit is set once the value at the distance array corresponding with the vertex is modified. This optimisation ensures that we only check edges which have a chance of being cheaper. If an edge has not been modified yet, i.e. the distance is still set to $\infty$, there is no chance it will provide a cheaper distance as $\infty + weight > \infty$.

It should be noted that we expect Bellman-Ford to be slower compared to MS-BFS. With Bellman-Ford, in the worst case, we have to perform $1 - |V|$, where $|V|$ is the number of vertices, iterations until we have found all the cheapest paths. With MS-BFS, we only have to iterate over all vertices once since we know the first time we discover a vertex, it is guaranteed to be the shortest path. With Bellman-Ford, we cannot give this guarantee after one iteration since it can be the case that a cheaper path is discovered later. This fact requires us to perform another iteration over all vertices. Only once all vertices have been iterated over, and no change has been made can we guarantee that all cheapest paths have been found. Even taking the best possible case for Bellman-Ford, where all cheapest

paths are found in the first iteration, we still require a second iteration over all vertices to ensure no more changes have to be made.

### 4.7.1 Vectorising batched Bellman-Ford

Batched Bellman-Ford, as shown in Algorithm 6 will not be auto-vectorised, as explained in Section 2.4 by the clang or GCC compilers[1]. This is due to the modified array and the if-statement at line 19 (`if newDist != dists[n][i] then`). This if-statement creates a branch, which does not necessarily prevent vectorisation (80), however, in this case, every lane (`i`) can be a different value, meaning that we wish to execute the code inside the if-statement only for some lanes. This if-statement makes vector operations difficult as we wish to do the same operation on all lanes in a vector.

Thus, we have tried to modify the algorithm such that it will be auto-vectorised. We do not need the modified array to guarantee the correct output, as this was an optimisation to avoid checking unnecessary lanes. Therefore, we have tried removing the modified array and the if-statement mentioned earlier. The result is shown in Algorithm 7. We have removed the code related to initialising all arrays as this is equal to what is shown in Algorithm 6.

**Algorithm 7** Batched Bellman-Ford Without The Modified Array

```
    bool changed = true
 2: while changed do
        changed = false
 4:     for each v in G.vertices do
            for each n in G.neighbours(v) do
 6:             int weight = edgeWeight(v,n)
                for i=0 to dists[v].size() do
 8:                 int minDist = min(dists[n][i], dists[v][i] + weight)
                    changed |= ((minDist < dists[n][i]) | changed)
10:                 dists[n][i] = minDist
                end for
12:         end for
        end for
14: end while
```

However, this will not generate vectorised instructions for the inner-most for-loop[2]. The

[1] `https://godbolt.org/z/zz7WeMshE`

[2] `https://gcc.godbolt.org/z/MfnMcb8he`

fact that no vectorised instructions are generated is likely due to the `changed` variable, which can be different for every lane in this case. Hence auto-vectorisation is not possible. We include this algorithm compared to the version, including the modified array.

In Algorithm 8 we show pseudocode that will be auto-vectorised by the GCC and clang compilers[1]. We have split the inner-most for-loop into functions. The function `update_one_lane(n_dist, v_dist, weight)` performs the same operations as was done previously, checking if the newly found path is cheaper than the current known one. If this is the case, we return a non-zero value; if not, we return zero. In the function `update_lane(dists, v, n)`, the for-loop is changed to a while loop.

In the current implementation, the edge weights are either 64-bit integers or doubles (also requiring 64 bits). However, in the case that the edge weights are integers, it could be that 64-bits are never fully utilised, and instead, a smaller amount of bits can be used per edge weight. If we use SSE instructions with 64-bit integers, it will only provide a speed-up of roughly 2x since the XMM registers are 128-bit wide; thus, $128/64 = 2$. Therefore, fewer bits per edge weight will provide a more significant speed-up.

[1]See all instructions: `https://godbolt.org/z/KWjKWPq51`

**Algorithm 8** Auto-Vectorised batched Bellman-Ford

```
Function update_one_lane(n_dist, v_dist, weight):
    int new_dist = v_dist + weight
    bool better = new_dist < n_dist
    int min = better ? new_dist : n_dist
    int diff = n_dist ⨁ min
    n_dist = min
    return diff
EndFunction

Function update_lane(dists, v, n):
    int weight = edgeWeight(v,n)
    int num_lanes = dists[v].size()
    int lane_idx = 0
    int xor_diff = 0
    while lane_idx < num_lanes do
        xor_diff |= update_one_lane(dists[n][lane_idx], dists[v][lane_idx], weight)
        ++lane_idx;
    end while
    return xor_diff != 0;
EndFunction

bool changed = true
while changed do
    changed = false
    for each v in G.vertices do
        for each n in G.neighbours(v) do
            changed = update_lanes(dists, v, n) | changed;
        end for
    end for
end while
```

# 5

# Experiments

## 5.1 Experimental Setup

To assess the scalability of the proposed algorithms, we conducted a number of experiments. These have been executed in the following environments:

- DuckDB version 0.2.2 Development
- Intel(R) Xeon(R) CPU E5-4657L v2 @ 2.40 GHz, 48 cores with Hyper-Threading
- 1 TiB RAM
- L1 cache: 1.5 MiB (data), 1.5 MiB (instr), L2 cache: 12 MiB, L3 cache: 120 MiB
- Compiler: GCC 9.4.0
- Operating system: Fedora release 32

For the microbenchmark discussed in section 5.4, the following environment was used:

- Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz, 4 cores with Hyper-Threading
- 16 GiB RAM
- L1 cache: 128 KiB (data), 128 KiB (instr), L2 cache: 1 MiB, L3 cache: 8 MiB
- Compiler: GCC 9.4.0
- Compiler options for SSE2: `-O3`
- Compiler options for AVX2: `-O3 -mavx2`
- Operating system: Ubuntu 20.04.4 LTS

### 5.1.1 LDBC Social Network Benchmark

The Linked Data Benchmark Council has created the Social Network Benchmark to test graph functionalities and performance of database management systems (81). The benchmark contains datasets structured similarly to real-world social networks, consisting

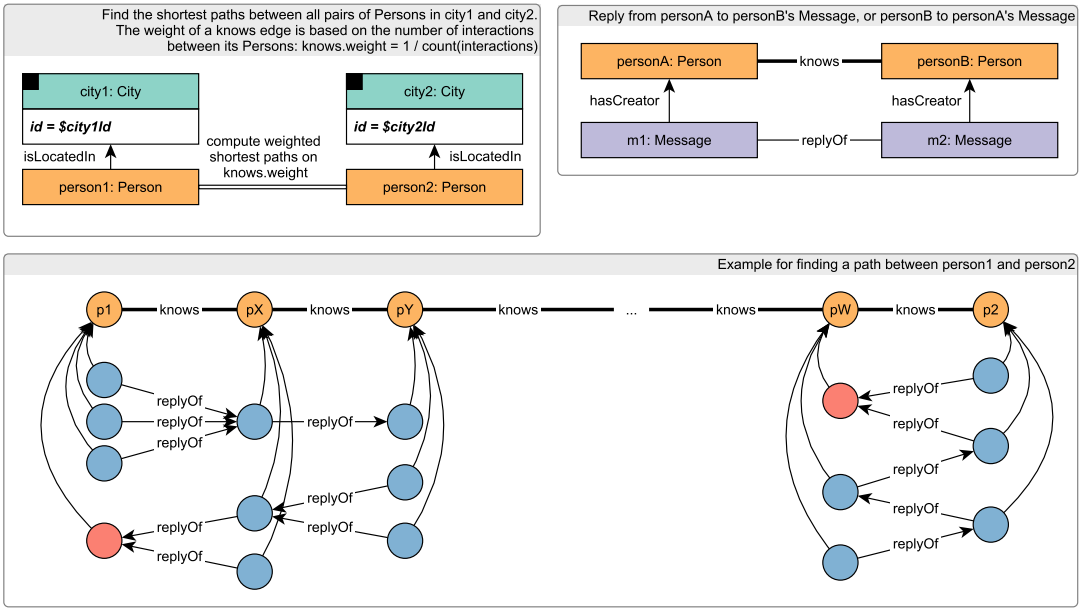



**Figure 5.1:** LDBC BI Query 19

of persons and their connections. These datasets exhibit natural social-network phenomena such as the small-world property (77). Two workloads are described in the benchmark, the *Interactive* workload, which focuses on transactional queries, and the *Business Intelligence* (BI) workload, which focuses on analytical queries.

The BI workload consists of 20 complex read queries and refresh operations (insert and delete operations). Since SQL/PGQ is a read-only query language, we will only be focusing on the queries related to the read operations. Each of the 20 queries contain substitution parameters 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 system's scalability. The more significant the scale factor, the more vertices and edges are contained in the graph.

Two queries require us to find the shortest or cheapest path from some starting point to a target. These queries are 19 (82), and 20 (83).

**Read Query 19.** For read query 19 (82), see Figure 5.1, the parameters are `City1` and `City2`. The goal is to find `Person1` and `Person2`, who are from their respective cities, where the interaction path is the cheapest. The interaction path is defined as the number of 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.

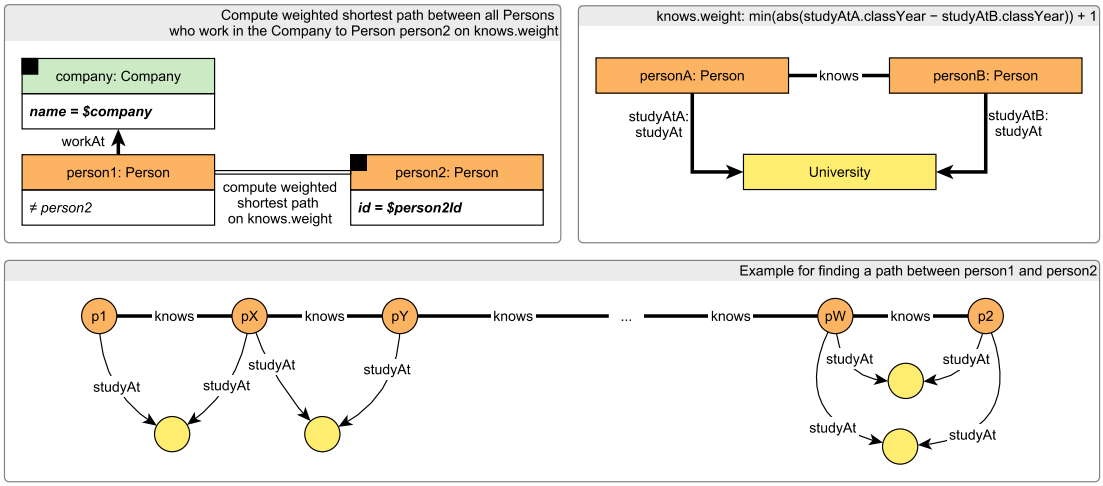



**Figure 5.2:** LDBC BI Query 20

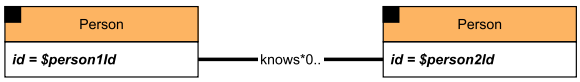



**Figure 5.3:** LDBC Interactive Query 13

**Read Query 20.** Read query 20 (83), see Figure 5.2, provides two parameters: `Person2` and `Company`. The goal is to find `Person1` who is working or has worked at `Company`, who has a path to `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 (to avoid division by zero errors). Since the weights between any two persons are not all equal, it is necessary to use the batched Bellman-Ford algorithm discussed in Section 4.7. In case all weights were equal, MS-BFS, discussed in Section 4.5 could be used.

**Interactive query 13** Interactive query 13, see Figure 5.3 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 the 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 of this query lies in its large search space. Unlike 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 consider all `knows` edges

provided in the dataset. Since the SNB emulates a social network with the small-world property (77), any two people will likely have a path containing six or fewer hops.

To create a direct comparison between the performance of our shortest path and cheapest path functions, we will also evaluate the cheapest path implementation on query 13. To do this, we assign a pseudo-random edge weight to every knows edge, namely `person1id + person2id % 10 + 1`.

The queries will be executed in several phases. In the first phase, we perform any pre-computing required for the query and try to filter the number of entries. By filtering, we reduce the search space, so fewer vertices and edges need to be traversed. The second phase consists of creating the CSR structure required for the shortest and cheapest path algorithms. The next step is to insert all parameters into a temporary table, as discussed in the previous section. It is then possible to compute the shortest or cheapest paths for all rows in this temporary table and return this.

**Disclaimer** The results shown in this thesis are not representative of an officially audited LDBC Benchmark, and therefore the results should not be interpreted as such.

## 5.2 Shortest Path

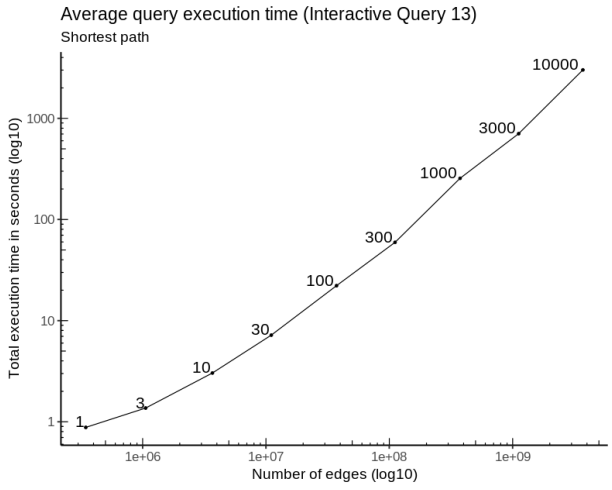



**Figure 5.4:** Average execution time per scale factor for shortest path Interactive query 13

In Figure 5.4 we observe the average run-time for Interactive query 13 using the shortest path function. This total execution time, seen on the y-axis, includes CSR creation time and the MS-BFS algorithm execution. The x-axis shows the number of edges that had to

be traversed by the MS-BFS algorithm. The numbers ranging from 1 to 10 000 indicate the scale factors. We observe a scaling that is close to linear with the number of edges. During this experiment, there have been 400 unique source-destination pairs for each scale factor for which we find the shortest path. The largest scale factor, 10 000, consists of three billion *knows* edges to traverse, which was done in 3013 seconds ($\approx$ 50 minutes).

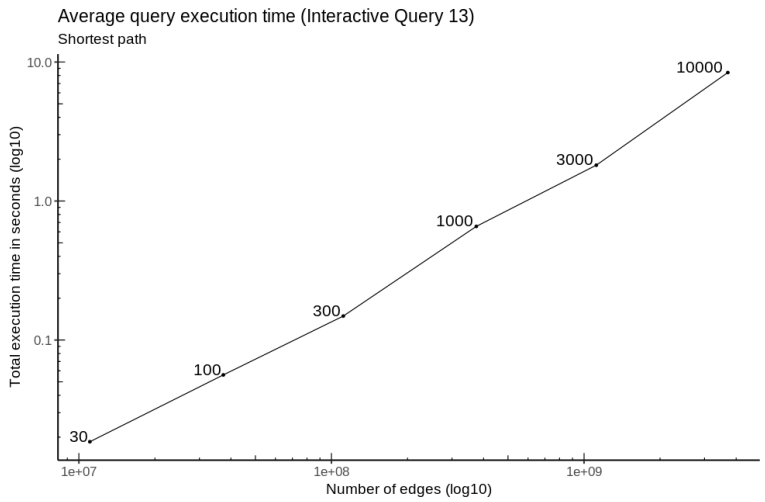



**Figure 5.5:** Average execution time per source-destination pair per scale factor for shortest path Interactive query 13

In Figure 5.5 we show the average run-time per source-destination pair. The y-axis is a logarithmic time scale with the total execution time in number of seconds. We observe that as the scale factor increases, the algorithm uses more time for every source-destination pair on average. For scale factor 10 000 it takes on average 10 seconds to find the shortest path for a source.

In Figure 5.6 we observe the relative time spent per execution phase of the shortest path query split up by the larger scale factors. Across all scale factors, most time is spent performing the path-finding algorithm. In the scale factors 30 up to 1 000, creating the CSR takes 5% of the time. However, in the most significant scale factors, it can be observed that CSR creation takes more time relatively compared to the smaller scale factors.

Figure 5.7 shows the total time taken to create the CSR data structure for the various scale factors. Note that the y-axis is a logarithmic scale. We observe that for the scale factors 1 to 30, the CSR is created in constant time, roughly 0.3 seconds. However, we see a linear increase in the time it takes to create this data structure for larger-scale factors.

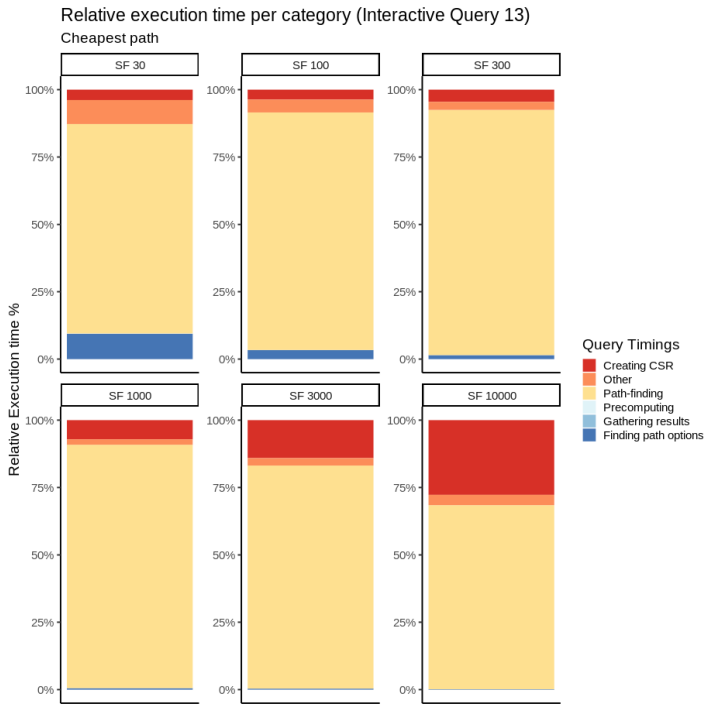



**Figure 5.6:** Relative time spent per phase for Interactive query 13 shortest path

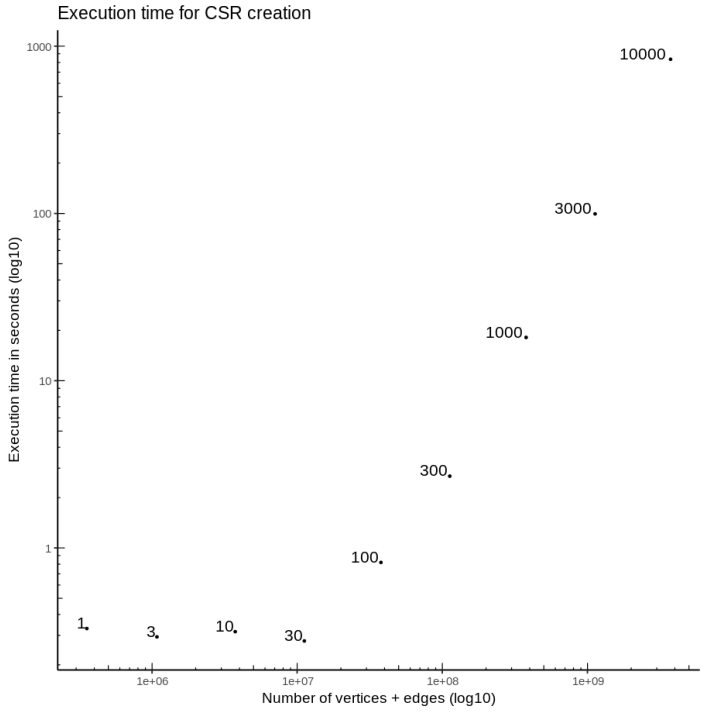



**Figure 5.7:** CSR creation time for all scale factors

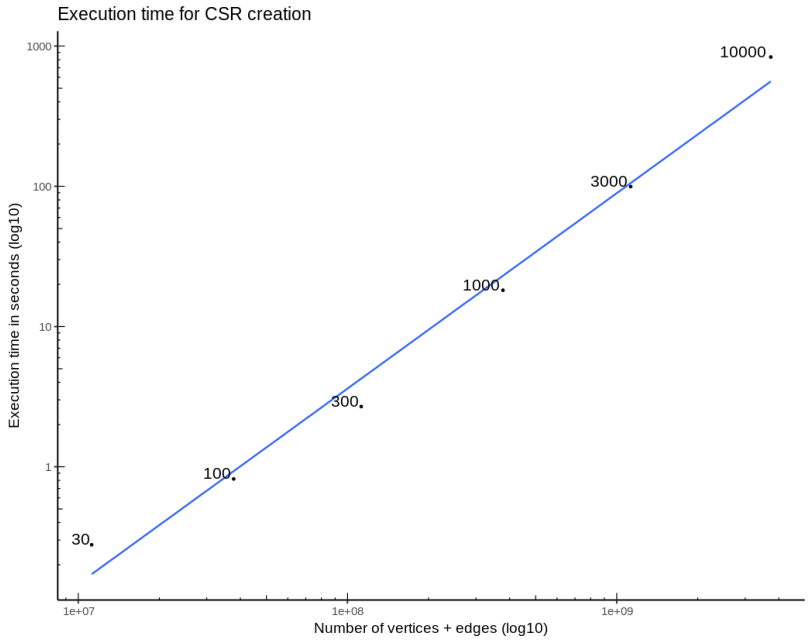



**Figure 5.8:** CSR creation time for the scale factors greater than or equal to 100

Figure 5.8 further highlights that the creation time increases linearly with the amount of edges and vertices for the larger scale factors.

## 5.3 Cheapest Path

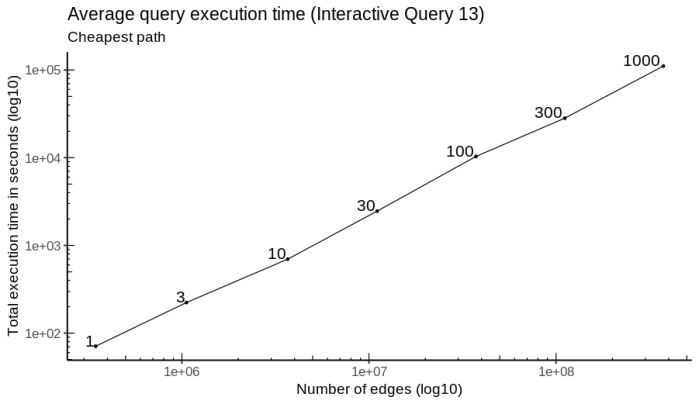



**Figure 5.9:** Total execution time per scale factor for cheapest path Interactive query 13

Figure 5.9 shows the total execution time for Interactive query 13. The y-axis is in seconds using a logarithmic scale. It can be observed that it is linearly scaling with the

number of edges. 1 000 is the highest scale factor which has been evaluated. Completing Interactive query 13 with 400 source-destinations pairs for scale factor 1 000 took 110 867 seconds, roughly 31 hours.

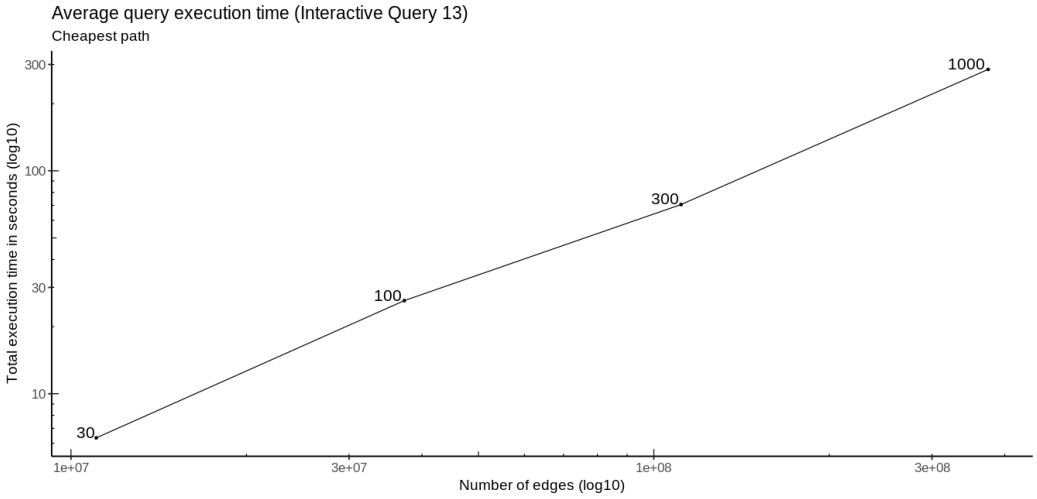



**Figure 5.10:** Average execution time per source-destination pair per scale factor for cheapest path Interactive query 13

Figure 5.10 shows the average time spent per source-destination pair for the most significant scale factors. For scale factor 30, on average less than 3 seconds is required for a source-destination pair. For scale factor 1 000, the cheapest path for a source-destination pair was found in 300 seconds on average.

Figure 5.11 shows the relative time spent in each phase of the query. As can be observed, the path-finding phase, in which batched Bellman-Ford is executed, dominates all other phases in terms of time spent.

## 5.4 Vectorized batched Bellman-Ford

To assess the performance of an auto-vectorized implementation of the batched Bellman-Ford algorithm, we have conducted a microbenchmark. The results of this benchmark can be observed in Figure 5.12a. We conduct a microbenchmark before integrating the auto-vectorized batched Bellman-Ford in DuckDB since it is not always guaranteed that vectorizing an algorithm improves the performance. The microbenchmark seen in Figure 5.12a is performed on a graph consisting of 1024 vertices, having an average of 100 edges each. This microbenchmark aims to keep the graph small so it can be contained

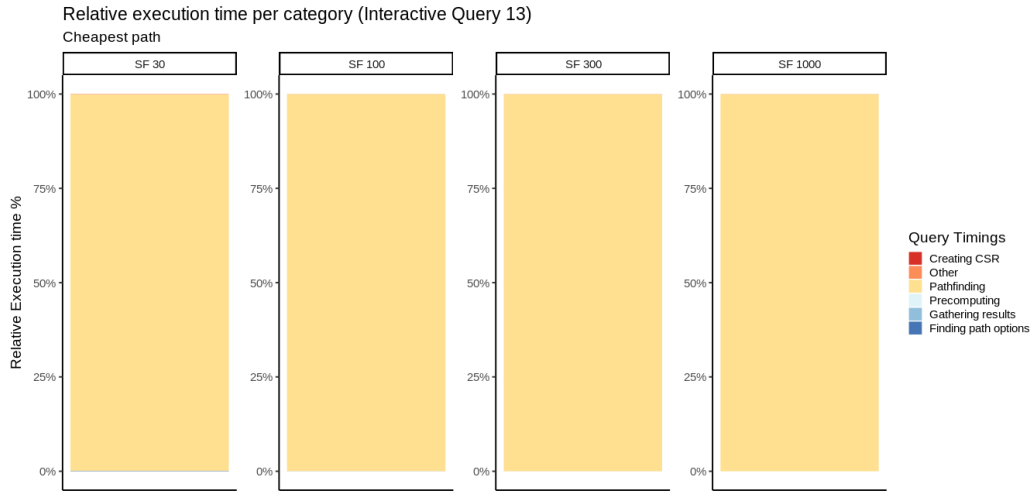
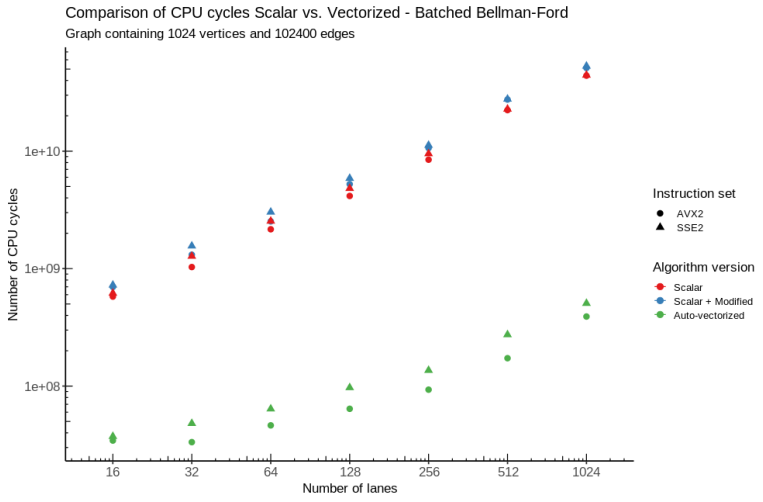



**Figure 5.11:** Relative time spent per phase for Interactive query 13 cheapest path




**(a)** Scalar vs. auto-vectorized implementations of batched Bellman-Ford

| Lanes | Speed-up (Scalar vs. AVX2) |
|---|---|
| 16 | 16× |
| 32 | 30× |
| 64 | 46× |
| 128 | 64× |
| 256 | 90× |
| 512 | 129× |
| 1 024 | 112× |

**(b)** Speed-up of scalar vs. AVX2 instructions for batched Bellman-Ford

**Figure 5.12:** Performance of scalar vs. auto-vectorized implementations of batched Bellman-Ford

in the cache. This allows for a more accurate assessment of the influence of the SIMD instructions compared to the scalar instructions.

In Figure 5.12a we observe on the y-axis a logarithmic scale on the number of CPU cycles taken to complete the execution of the batched Bellman-Ford algorithm. On the x-axis, we see the number of lanes (each representing a unique source).

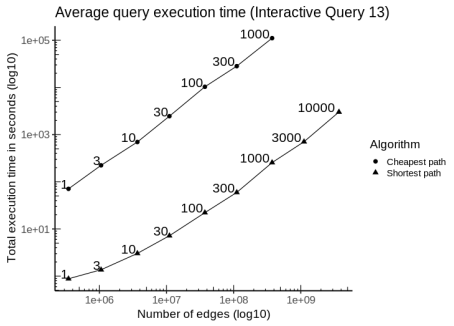
We have compared three implementations of the batched Bellman-Ford algorithm. The first one is the scalar implementation without any optimizations. The second one is also a scalar implementation with the added optimization of the modified array used to avoid checking unnecessary neighbours. The third is the auto-vectorized implementation. We have also performed experiments on both the SSE2 and AVX2 instruction sets.

As shown in Figure 5.12a, the auto-vectorized implementation is faster than both scalar implementations across all lanes. Since the y-axis is a logarithmic scale, direct comparisons can be difficult. Thus we have included Figure 5.12b to highlight the speedup across the various lanes between the scalar and the vectorized AVX2 implementation. We observe that when we have 16 lanes, the vectorized AVX2 version is 16 times faster. The most considerable speedup is achieved when there are 512 lanes, in which case the vectorized version is 129 times faster. The speedup is lower when doubling the number of lanes to 1024, in which case the speedup is 112x.

# 6

# Discussion

## 6.1 Performance comparison between shortest and cheapest path functions




**(a)** Difference in performance between shortest and cheapest path for query 13

| Scale factor | Speed-up (Cheapest vs. shortest) |
|---|---|
| 1 | 80× |
| 3 | 163× |
| 10 | 230× |
| 30 | 342× |
| 100 | 466× |
| 300 | 475× |
| 1 000 | 434× |
| 3 000 | N/A |
| 10 000 | N/A |

**(b)** Speed-up of cheapest path function / shortest path function for query 13

**Figure 6.1:** Difference in performance between shortest and cheapest path for query 13

The performance gap between the shortest and cheapest path functions appears to be several orders of magnitude, see Figure 6.1a. Since the y-axis uses a logarithmic scale, Figure 6.1b has been included to make comparison easier. The shortest path function is at most 475× faster. MS-BFS requires a single iteration over all vertices, while Bellman-Ford requires potentially $|V| - 1$ iterations, where $|V|$ is the number of vertices. If the graph is larger than can be stored in memory, this causes additional cache misses every iteration. The cache sizes of the machine tested are:

- L1: 1.5 MiB (data), 1.5 MiB (instr)
- L2: 12 MiB

- L3: 120 MiB

The CSR structure in DuckDB uses 64-bit integers to store the vertices, edges, and weights in the case of a weighted graph. In scale factor 1, there are 9538 vertices and 346028 edges. Thus the CSR structure requires $9538 \cdot 64 = 76$ KiB for the vertices, $346\,028 \cdot 64 = 2768$ KiB $= 2.7$ MiB for the edges, and $346\,028 \cdot 64 = 2768$ KiB $= 2.7$ MiB for the weights. Thus, all CSR data can be stored in the L2 cache for the smallest scale factors.

For the shortest path function, there are three bitmaps of length $|V|$ which take $3 \cdot 9538 \cdot 400 = 1430$ KiB total space. Additionally, the shortest distance is stored using 8-bit integers initially [1]. For every source, a new lane with length $|V|$ is created. Therefore in query 13, the distance matrix with 400 unique sources required $400 \cdot 9538 \cdot 8 = 3815$ KiB. Therefore, the total size for the shortest path function is $3815 + 1430 + 2768 + 76 = 8089$ KiB $= 7.9$ MiB. This data all fits within the L2 cache.

The distances are stored in a 64-bit integer matrix for the cheapest path. For every source, a new lane is created of length $|V|$. When testing with 400 unique sources, this matrix has a total size of $400 \cdot 64 \cdot 9538 = 30\,521$ KiB $= 30$ MiB. Additionally, there is a bitmap of length $|V|$ to keep track of the modified entries, which takes $400 \cdot 9538 = 477$ KiB. The total size required for computing the cheapest path is: $477 + 30521 + 2768 + 2768 + 76 = 36\,610$ KiB $= 35.7$ MiB. This data is more than can be stored in the L2 cache; therefore, L3 is required. Since batched Bellman-Ford requires multiple iterations over all vertices, this will cause cache misses and cause a slow-down.

For more significant scale factors, the shortest path function will also be unable to store all data in L2 caches. However, the growth in total size is faster for the cheapest path because the distances are stored in 64-bit integers, as opposed to 8-bit integers for the shortest path function. In addition, the CSR data structure requires both the edge array and the equally sized weight array for the cheapest path, whereas only the edge array is required for the shortest path.

Due to all these factors, it is expected that batched Bellman-Ford is slower than MS-BFS. In Chapter 7 optimisations will be covered to reduce the total size used for computing the cheapest path. Reducing the size will allow more data for computing the cheapest path to be stored in the cache and reduce the number of cache misses.

[1] We assume no overflows occurred in this test, which would necessitate scaling up us to scale up to 16-bit integers to store the distance

## 6.2 Vectorising batched Bellman-Ford

The microbenchmark was conducted using 32-bit integers for all components. The components include the CSR vertex array, the CSR edge array, and the distances matrix in the batched Bellman-Ford algorithm. The graph contained a total of 1024 vertices, thus the CSR vertex array used $1024 \cdot 32 = 4$ KiB. Each vertex has 100 outgoing edges, thus $1024 \cdot 32 \cdot 100 = 400$ KiB. The distance array, with length $|V|$, grows with the number of sources or lanes. The lowest amount of lanes tested was 16. Thus the array has a total size of $1024 \cdot 32 \cdot 16 = 64$ KiB. The widest lane, 1024, had a total size of $1024 \cdot 32 \cdot 1024 = 4.1$ MiB.

The cache sizes of the machine tested are:

- L1: 128 KiB (data), 128 KiB (instr)
- L2: 1 MiB
- L3: 8 MiB

Thus, in the smallest case the total size is $4 + 400 + 64 = 468$ KiB. Therefore, not all data could have been stored in L1. However, all data did fit in L2. For the largest number of lanes, the total size was $4 + 400 + 4198 = 4602$ KiB $= 4.6$ MiB. This data is too large for L2; thus, the L3 cache must be used. The threshold where the L3 has to be used is exceeded at 256 lanes ($1024 \cdot 32 \cdot 256 = 1.048$ MiB).

The speed-up shown in Figure 5.12b is more significant than was expected solely from SIMD instructions. When using AVX2 instruction, it is possible to handle $256/32 = 8$ lanes at a time. Thus a speed-up of 8x was expected. However, this speed-up is already more significant in the narrowest amount of lanes.

Another benefit is the amortisation that occurs when handling more lanes at a time. When there are more lanes, it can be that the average amount of active lanes increases. If only a single lane is active, it is expensive to visit the edges, as this is a random memory access. However, with the higher number of lanes, the cost of visiting the edges gets amortised. The added benefit is that the calculations use vectorised instructions and the computation is only being done once for every 8 lanes in this case.

# 7

# Future Work

In this chapter, we discuss potential performance optimisations for the algorithms presented in this thesis.

## 7.1 Path-finding optimisations

An optimisation for the MS-BFS algorithm, see Section 4.5, is to perform a bidirectional search. In this optimisation, exploration would be started from both the source and destination vertices to discover a common vertex. In the current implementation, the exploration is started only from the source vertices.

Since every search is an expensive operation, it is crucial to improve the performance to minimise the number of searches. Therefore, searches should only be executed on unique sources. Within a single vector, this can be done by only creating a lane for every unique source. However, the case where we have to handle multiple vectors is more likely, in which case a vector is not aware of the sources in any other vector. The possibility then exists that a path-finding search is done on the same sources in different vectors.

To work around this limitation, the optimisation would store the already computed distances for a given source in the client context. The client context is also where the CSR data structure is stored. The structure would be a map in which the key corresponds with the source vertex's row identifier. The value would be an array with the length of all vertices where the distances from the source to all other vertices are stored. Since vectors can be executed in parallel, inserting these values would have to be an atomic operation. Once the distance is stored, future vectors can check if the distances from a source have already been computed. If this is the case, there is no need to recompute the distances.

A similar optimisation to reduce the number of searches is to detect when the query is a multi-source single-destination. In this case, it is better to reverse the CSR order, with the sources and destinations swapped. It is then possible to start the path-finding search from a single destination and discover the distances to the multi-sources.

An alternative to the batched Bellman-Ford is to implement bidirectional Dijkstra's algorithm. A critical difference between Bellman-Ford and Dijkstra is that the latter finds the cheapest path first. In comparison, Bellman-Ford finds longer, cheaper paths in every iteration. Only in the final iteration can it be guaranteed that the cheapest path has been found in Bellman-Ford. This fact makes it impossible to use a bidirectional implementation in which a search is started from both the source and the destination vertices. However, in Dijkstra's algorithm, it is possible to perform a bidirectional search since the cheapest path is found first. Similarly to bidirectional MS-BFS, with bidirectional Dijkstra, a search is started from both the source and the destination. It continues until a common vertex has been found. The cheapest distance between a source and destination is found at this point.

## 7.2 CSR optimisations

An optimisation related to the vectorised batched Bellman-Ford algorithm, see Section 4.7 starts at the CSR creation. The optimisation would keep track of the maximum weight of an edge. When the maximum edge weight is known, it is possible to minimise the data type used to store the distances for the Bellman-Ford algorithm. Variations exist, and experiments should be conducted to test the effectiveness of these variations. One variation would be to multiply the maximum edge weight with the number of edges. It is then guaranteed that no cheapest path can exceed this value. If this value is below, for example, the maximum value that can be stored in an 8-bit integer, it is safe to use 8-bit integers. Using fewer bits allows us to fill more lanes in a single SIMD register and thus use fewer instructions overall. However, as the number of edges can quickly become large, this could quickly default to a 64-bit integer.

Since the number of hops is often tiny, given the small-world network property (77), it is possible to multiply the maximum edge weight by the likely maximum number of hops. If this value is under, for example, the maximum 8-bit integer, the algorithm can be started using 8-bit integers to store the distances. If an overflow can occur, the 8-bit distances can be copied to 16-bit integers. Such an overflow-prevention mechanism has already been implemented for the MS-BFS implementation.

This optimisation aims to minimise the data type used to store the distances. By minimising the data type, we can fill the SIMD registers with more lanes and thus handle more lanes with fewer instructions.

## 7.3 Miscellaneous improvements

The limited parallelisation while performing the searches is a limitation of the current approach. In DuckDB, the number of threads used for a query is decided during the table scan. For approximately every 120 000 rows, a new thread is used. However, as shown in the experiments, the path-finding queries will not often contain more than 120 000 rows in a realistic setting. Furthermore, in the queries tested, the number of source-destination pairs is much smaller than the number of rows in the table. This fact causes most vectors to be close to empty, in the worst case, only having a single element. A solution could be introducing a new operator that ensures the vectors are as complete as possible. It uses as input the emptier vectors and waits until it has a full vector before continuing. By ensuring that the vectors are full, we can utilise the lanes better.

# 8

# Conclusion

This thesis aimed to create efficient path-finding operators for SQL/PGQ. The path-finding operators have been implemented in the open-source RDBMS DuckDB. Several research questions have been formulated in Section 1.2 which will be answered in this conclusion.

**How to best implement path-finding algorithms in DuckDB?** This research question contained two sub-questions, as graphs defined in SQL/PGQ can either be weighted or unweighted. The algorithms have been implemented as scalar UDFs in DuckDB, allowing for the parallelisation to be handled by DuckDB internally.

An important aspect is the avoidance of cache misses, reducing CPU efficiency. Therefore, a CSR data structure is created that represents the graph. This CSR can be created in linear time for any size graph, as the implementation does not rely on sorting. For larger graphs, the time taken to create the CSR scales linearly with the number of edges and vertices.

We have found batched path-finding algorithms to be a scalable solution for both weighted and unweighted graphs.

Path-finding for unweighted graphs was implemented using the MS-BFS algorithm published by Then et al. (75). This implementation can be further optimised using a bidirectional search.

The path-finding for weighted graphs relies on batched Bellman-Ford, also published by Then et al. (79). As expected, batched Bellman-Ford has shown to be slower than MS-BFS. However, it also shows to scale linearly.

As an optimisation for the batched Bellman-Ford, we have looked at creating a vectorised batched Bellman-Ford implementation. A microbenchmark showed that this could potentially improve the performance of the batched Bellman-Ford by two orders of magnitude, reducing the performance gap to the MS-BFS implementation.

# 8. CONCLUSION

To conclude, DuckDB has shown to be an ideal candidate for creating a SQL/PGQ implementation through the use of scalar UDFs and the selected algorithms fit well into DuckDB's execution model.

# References


[1] Bin Shao, Yatao Li, Haixun Wang, and Huanhuan Xia. **Trinity Graph Engine and its Applications**. *IEEE Data Eng. Bull.*, **40**(3):18–29, 2017. 1

[2] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. **The ubiquity of large graphs and surprising challenges of graph processing: extended survey**. *VLDB J.*, **29**(2-3):595–618, 2020. 1, 23

[3] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. **The Case Against Specialized Graph Analytics Engines**. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015. 1, 25

[4] M. Tamer Özsu. **Graph Processing: A Panoramic View and Some Open Problems**. In *VLDB*, 2019. 1, 7

[5] Jan Michels and Andy Witkowski. **Graph database applications with SQL/PGQ**. `https://download.oracle.com/otndocs/products/spatial/pdf/AnD2020/AD_Develop_Graph_Apps_SQL_PGQ.pdf`, 2020. Accessed: 08/02/2022. 1, 3, 10, 12

[6] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. **Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries**. *CoRR*, **abs/1910.09017**, 2019. 1, 5, 6, 23

[7] TigerGraph. **TigerGraphDB**. `https://www.tigergraph.com/`, 2022. Accessed: 02/02/2022. 1, 6

[8] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. **Cypher: An Evolving Query Language for Property Graphs**. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1433–1445. ACM, 2018. 1, 8

[9] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. **PGQL: A Property Graph Query Language**. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, New York, NY, USA, 2016. Association for Computing Machinery. 1, 8, 9, 10

[10] ISO/IEC JTC 1/SC 32 Data management and interchange. **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`, 2022. Accessed: 17/02/2022. 2, 3

[11] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. **Foundations of Modern Query Languages for Graph Databases**. *ACM Comput. Surv.*, **50**(5):68:1–68:40, 2017. 2, 8, 10

[12] Guodong Jin, Nafisa Anzum, and Semih Salihoglu. **GrainDB: A Relational-core Graph-Relational DBMS**. In *CIDR*, 2022. 2, 8, 10, 25

[13] DuckDB Contributors. **DuckDB Documentation - WITH Clause**. `https://duckdb.org/docs/sql/query_syntax/with`, 2022. Accessed: 14/03/2022. 2

[14] Jim Melton and Alan R. Simon. **Sql: 1999 Understanding Relational Language Components**. 2002. 3

[15] 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. **G-CORE: A Core for Future Graph Query Languages**. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD*

*Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1421–1432. ACM, 2018. 3, 8, 9

[16] 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, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. **Graph Pattern Matching in GQL and SQL/PGQ**. In Zachary Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 2246–2258. ACM, 2022. 3, 9, 11, 12

[17] Alastair Green. **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`, 2018. 3

[18] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. **LSQB: a large-scale subgraph query benchmark**. In Vasiliki Kalavri and Nikolay Yakovets, editors, *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*, pages 8:1–8:11. ACM, 2021. 3

[19] ISO/IEC JTC 1/SC 32 Data management and interchange. **ISO/IEC CD 39075: Information Technology — Database Languages — GQL**. `https://www.iso.org/standard/76120.html`, 2022. Accessed: 17/02/2022. 3

[20] Mark Raasveldt and Hannes Mühleisen. **DuckDB: an Embeddable Analytical Database**. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1981–1984. ACM, 2019. 3, 30, 31

[21] DuckDB Labs. **New CWI spin-off company DuckDB Labs: Solutions for fast database analytics**. `https://duckdblabs.com/news/spin-off-company-DuckDB-Labs/`, 2021. Accessed: 02/02/2022. 3

[22] Tavneet Singh, Gábor Szárnyas, and Peter Boncz. **Integrating SQL/PGQ to DuckDB**. https://docs.google.com/presentation/d/1aeHCOuopl8r7Lk-TtqpYSEsXblni632rJoVUR62jc7s, 2022. Accessed: 03/02/2022. 3, 32

[23] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases, 2nd Edition*. O'Reilly Media, Inc., 2015. 6

[24] Oracle Labs. **Oracle PGX**. https://docs.oracle.com/cd/E56133_01/latest/index.html, 2022. Accessed: 08/03/2022. 6

[25] Guus Schreiber and Yves Raimond. **W3C RDF 1.1 Primer**. https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/, 2014. Accessed: 11/03/2022. 6

[26] 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. **Knowledge Graphs**. *ACM Comput. Surv.*, **54**(4), jul 2021. 6

[27] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. **SQLGraph: An Efficient Relational-Based Property Graph Store**. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1887–1901. ACM, 2015. 6, 21

[28] BlazeGraph. **BlazeGraphDB**. https://blazegraph.com/, 2022. Accessed: 02/02/2022. 7

[29] Bradley R. Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, Simone Rondelli, Alexander Ryazanov, Michael Schmidt, Kunal Sengupta, Bryan B. Thompson, Divij Vaidya, and Shawn Wang. **Amazon Neptune: Graph Data Management in the Cloud**. In Marieke van Erp, Medha Atre, Vanessa López, Kavitha Srinivas, and Carolina Fortuna, editors, *Proceedings of the ISWC 2018 Posters &*

*Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, October 8th - to - 12th, 2018*, **2180** of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018. 7

[30] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. **Building an efficient RDF store over a relational database**. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 121–132. ACM, 2013. 7

[31] Thomas Neumann and Gerhard Weikum. **The RDF-3X engine for scalable management of RDF data**. *VLDB J.*, **19**(1):91–113, 2010. 7

[32] Gabor Szarnyas. **Self-joins**. `https://szarnyasg.github.io/posts/self-joins/`, 2022. Accessed: 11/03/2022. 7

[33] Neo4j. **Neo4j**. `https://neo4j.com/`, 2022. Accessed: 29/06/2022. 8

[34] SAP. **Work With Graph Workspaces in SAP HANA Database Explorer**. `https://help.sap.com/docs/SAP_HANA_PLATFORM/f381aa9c4b99457fb3c6b53a2fd29c02/57e1cbf376064b2aba49f455269f8202.html?version=2.0.03&locale=en-US`, 2022. Accessed: 06/06/2022. 8

[35] Pieter Cailliau, Tim Davis, Vijay Gadepally, Jeremy Kepner, Roi Lipman, Jeffrey Lovitz, and Keren Ouaknine. **RedisGraph GraphBLAS Enabled Graph Database**. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 285–286. IEEE, 2019. 8

[36] Dhiman Sarma, Wahidul Alam, Ishita Saha, Mohammad Nazmul Alam, Mohammad Jahangir Alam, and Sohrab Hossain. **Bank Fraud Detection using Community Detection Algorithm**. In *2020 Second International Conference on Inventive Research in Computing Applications (ICIRCA)*, pages 642–646, 2020. 8

[37] Benjamin A. Steer, Alhamza Alnaimi, Marco A. B. F. G. Lotz, Félix Cuadrado, Luis M. Vaquero, and Joan Varvenne. **Cytosm: Declarative**

**Property Graph Queries Without Data Migration**. In Peter A. Boncz and Josep Lluís Larriba-Pey, editors, *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*, pages 4:1–4:6. ACM, 2017. 8, 21

[38] Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Martin Schuster, Petra Selmer, and Hannes Voigt. **Updating Graph Databases with Cypher**. *Proc. VLDB Endow.*, **12**(12):2242–2253, 2019. 9

[39] Peter Boncz. **A Survey Of Current Property Graph Query Languages**. `https://homepages.cwi.nl/~boncz/job/gql-survey.pdf`, 2022. Accessed: 17/02/2022. 10

[40] Neo4j Query Languages Standards and Research Team. **GQL Scope and Features**. 12 2018. 10

[41] Michael L. Fredman and Robert Endre Tarjan. **Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms**. *J. ACM*, **34**(3):596–615, jul 1987. 16

[42] Michael J. Bannister and David Eppstein. **Randomized Speedup of the Bellman-Ford Algorithm**, 2011. 16

[43] Jin Y. Yen. **An algorithm for finding shortest routes from all source nodes to a given destination in general networks**. *Quarterly of Applied Mathematics*, **27**:526–530, 1970. 16, 48

[44] Intel. **Intel Launches the Pentium® III Processor**. `https://www.intel.com/pressroom/archive/releases/1999/dp022699.htm`, 1999. Accessed: 13/07/2022. 18

[45] Bob Bentley. **Validating the Intel Pentium 4 Microprocessor**. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 244–248. ACM, 2001. 18

[46] Steam. **Steam Hardware and Software Survey**. `https://store.steampowered.com/hwsurvey/`, 2022. Accessed: 05/07/2022. 19

[47] David Kanter. **Intel's Sandy Bridge Microarchitecture**. `https://www.realworldtech.com/sandy-bridge/6/`, 2010. Accessed: 13/07/2022. 19

[48] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 1.* Intel Corporation, August 2007. 19

[49] Intel. **Intel® Intrinsics Guide**. `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`, 2022. Accessed: 13/07/2022. 19, 39

[50] Google. **cpu_features**. `https://github.com/google/cpu_features`, 2022. Accessed: 13/07/2022. 20

[51] Areej Syed. **Difference Between L1, L2, and L3 Cache: How Does CPU Cache Work?** `https://www.hardwaretimes.com/difference-between-l1-l2-and-l3-cache-how-does-cpu-cache-work/`, 2022. Accessed: 29/07/2022. 20

[52] Konstantinos Xirogiannopoulos, Virinchi Srinivas, and Amol Deshpande. **GraphGen: Adaptive Graph Processing using Relational Databases**. In Peter A. Boncz and Josep Lluís Larriba-Pey, editors, *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*, pages 9:1–9:7. ACM, 2017. 21, 22

[53] Kangfei Zhao and Jeffrey Xu Yu. **Graph Processing in RDBMSs**. *IEEE Data Eng. Bull.*, **40**(3):6–17, 2017. 22

[54] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Suijun Tong, Wen Sun, Thomas Kolanko, Md. Shahidul Haque Apu, and Huijuan Peng. **IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2**. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 345–359. ACM, 2020. 22

[55] Harsh Thakkar, Dharmen Punjani, Jens Lehmann, and Sören Auer. **Two for one: querying property graph databases using SPARQL via gremlinator**. In Akhil Arora, Arnab Bhattacharya, George H. L. Fletcher, Josep Lluís Larriba-Pey, Shourya Roy, and Robert West, editors, *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph*

*Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Houston, TX, USA, June 10, 2018*, pages 12:1–12:5. ACM, 2018. 22

[56] Eric Prud'hommeaux and Andy Seaborne. **SPARQL Query Language for RDF**. `https://www.w3.org/TR/rdf-sparql-query/`, 2022. Accessed: 07/07/2022. 22

[57] Harsh Thakkar, Renzo Angles, Marko Rodriguez, Stephen Mallette, and Jens Lehmann. **Let's build Bridges, not Walls: SPARQL Querying of TinkerPop Graph Databases with Sparql-Gremlin**. In *IEEE 14th International Conference on Semantic Computing, ICSC 2020, San Diego, CA, USA, February 3-5, 2020*, pages 408–415. IEEE, 2020. 22

[58] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. **The future is big graphs: a community view on graph processing systems**. *Commun. ACM*, **64**(9):62–71, 2021. 23

[59] Martin Junghanns, Max Kiessling, Niklas Teichmann, Kevin Gómez, André Petermann, and Erhard Rahm. **Declarative and Distributed Graph Analytics with GRADOOP**. *Proc. VLDB Endow.*, **11**(12):2006–2009, aug 2018. 24

[60] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. **Pregel: a system for large-scale graph processing**. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010. 24, 25

[61] The Apache Software Foundation. **Apache Spark**. `https://spark.apache.org/`, 2022. Accessed: 22/06/2022. 24

[62] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. **GraphLab: A New Framework For Parallel Machine Learning**. *CoRR*, **abs/1408.2041**, 2014. 25

[63] Domagoj Vrgoc, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. **MillenniumDB: A Persistent, Open-Source, Graph Database**. *CoRR*, **abs/2111.01540**, 2021. 26

[64] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. **Worst-case Optimal Join Algorithms**. *J. ACM*, **65**(3):16:1–16:40, 2018. 26

[65] Denny Vrandecic and Markus Krötzsch. **Wikidata: a free collaborative knowledgebase**. *Commun. ACM*, **57**(10):78–85, 2014. 26

[66] Jena Team. **Jena TDB Documentation**. `https://jena.apache.org/documentation/tdb/`, 2022. Accessed: 23/06/2022. 26

[67] Bryan B. Thompson, Mike Personick, and Martyn Cutcher. **The Bigdata® RDF Graph Database**. In Andreas Harth, Katja Hose, and Ralf Schenkel, editors, *Linked Data Management*, pages 193–237. Chapman and Hall/CRC, 2014. 26

[68] Orri Erling. **Virtuoso, a Hybrid RDBMS/Graph Column Store**. *IEEE Data Eng. Bull.*, **35**(1):3–8, 2012. 26

[69] Jim Webber. **A programmatic introduction to Neo4j**. In Gary T. Leavens, editor, *SPLASH'12 - Proceedings of the 2012 ACM Conference on Systems, Programming, and Applications: Software for Humanity, Tucson, AZ, USA, October 21-25, 2012*, pages 217–218. ACM, 2012. 27

[70] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009. 30

[71] Peter A. Boncz, Marcin Zukowski, and Niels Nes. **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*, pages 225–237. www.cidrdb.org, 2005. 31

[72] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. **Column-stores vs. row-stores: how different are they really?** In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 967–980. ACM, 2008. 31

[73] Laurens Kuiper. **Fastest table sort in the West - Redesigning DuckDB's sort**. `https://duckdb.org/2021/08/27/external-sorting.html`, 2021. Accessed: 04/05/2022. 32

[74] DuckDB Labs. **DuckDB - Testing**. `https://duckdb.org/dev/testing`, 2022. Accessed 19/03/2022. 32

[75] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. **The More the Merrier: Efficient Multi-Source Graph Traversal**. *Proc. VLDB Endow.*, **8**(4):449–460, dec 2014. 33, 37, 39, 71

[76] Thomas Vilhena. **Index-free adjacency**. `https://thomasvilhena.com/2019/08/index-free-adjacency`, 2019. Accessed: 21/02/2022. 35

[77] D. J. Watts and S. H. Strogatz. **Collective dynamics of 'small-world' networks**. *Nature*, (393):440–442, 1998. 39, 54, 56, 68

[78] DuckDB Labs. **DuckDB - List**. `https://duckdb.org/docs/sql/data_types/list`, 2022. Accessed: 01/07/2022. 46

[79] Manuel Then, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. **Efficient Batched Distance, Closeness and Betweenness Centrality Computation in Unweighted and Weighted Graphs**. *Datenbank-Spektrum*, **17**(2):169–182, 2017. 46, 47, 71

[80] Stefanos Baziotis. **A Beginner's Guide to Vectorization By Hand: Part 3**. `https://baziotis.cs.illinois.edu/performance/a-beginners-guide-to-vectorization-by-hand-part-3.html`, 2021. Accessed: 19/07/2022. 49

[81] 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. **The LDBC Social Network Benchmark**. *CoRR*, **abs/2001.02299**, 2020. 53

[82] Linked Data Benchmark Council. **LDBC SNB BI read 19**. `https://ldbcouncil.org/ldbc_snb_docs/bi-read-19.pdf`, 2022. Accessed: 13/06/2022. 54

[83] Linked Data Benchmark Council. **LDBC SNB BI read 20**. `https://ldbcouncil.org/ldbc_snb_docs/bi-read-20.pdf`, 2022. Accessed: 13/06/2022. 54, 55

# Appendix A

# SQL vs SQL/PGQ queries

```
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 A.1:** SQL/PGQ query transformed into SQL query

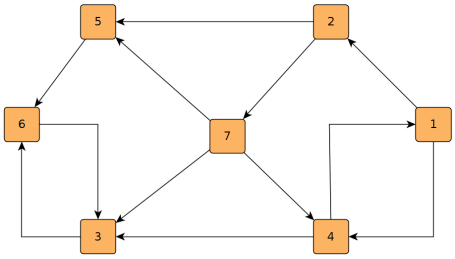
```
select gt.c1id, gt.c2id
from GRAPH_TABLE (aml, MATCH (c1 IS CUSTOMER)-[t1 IS TRANSFERS*]->(c2 IS
    CUSTOMER) COLUMNS (c1.cid AS c1id, c2.cid AS c2id)) gt

/* 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)
);
```

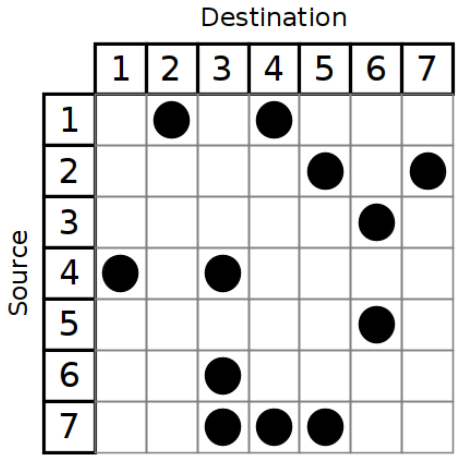
**Listing A.2:** SQL/PGQ query transformed into SQL query with Kleene star

# Appendix B

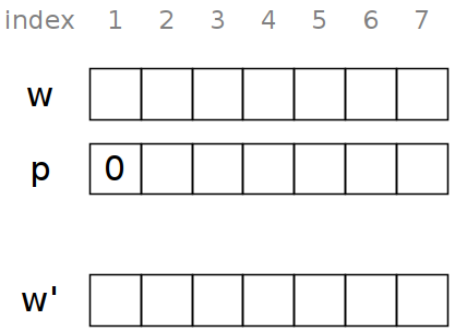
# Any shortest path example




**(a)** Graph representation of initial state

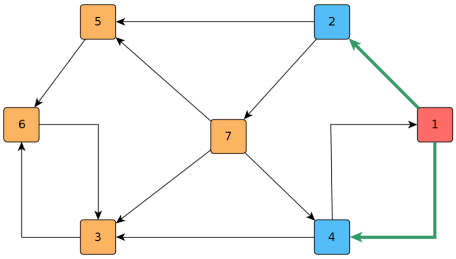



**(b)** Matrix representation of initial state

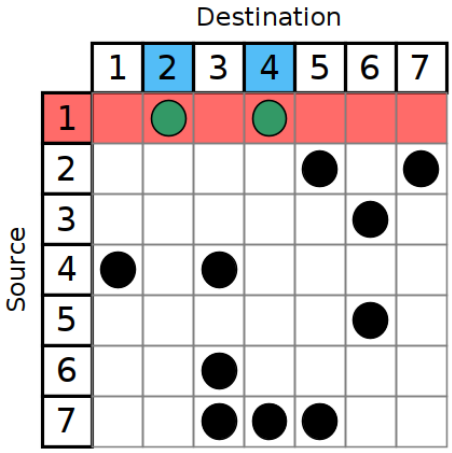



**(c)** Initial arrays

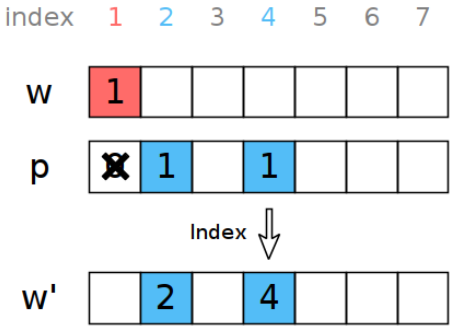
**Figure B.1:** Any shortest path initial state




**(a)** Graph representation of step 1

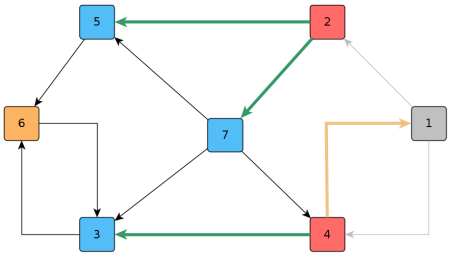



**(b)** Matrix representation of step 1

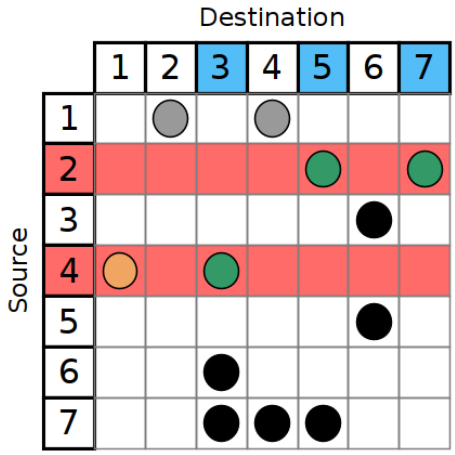



**(c)** Arrays after step 1

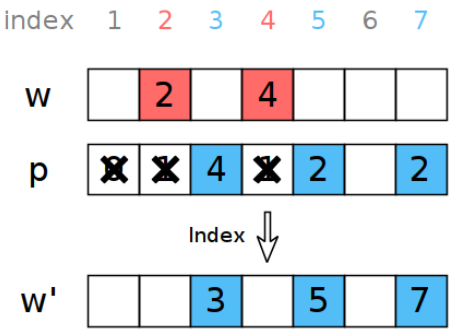
**Figure B.2:** Any shortest path step 1




**(a)** Graph representation of step 2

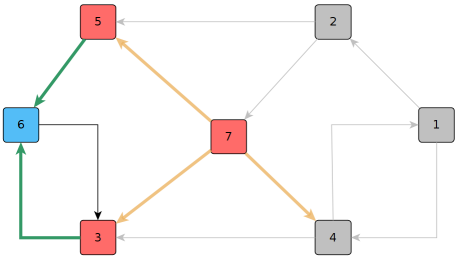



**(b)** Matrix representation of step 2

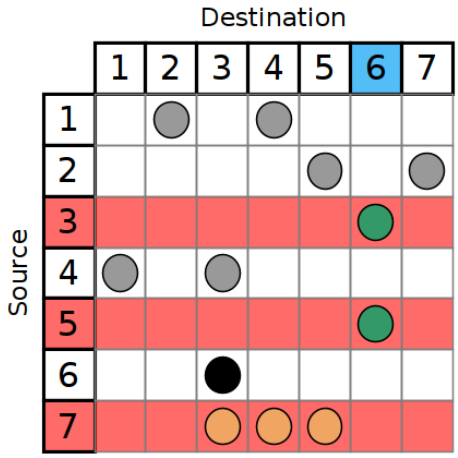



**(c)** Arrays after step 2

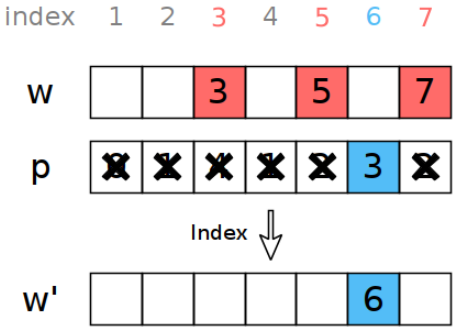
**Figure B.3:** Any shortest path step 2




**(a)** Graph representation of step 3

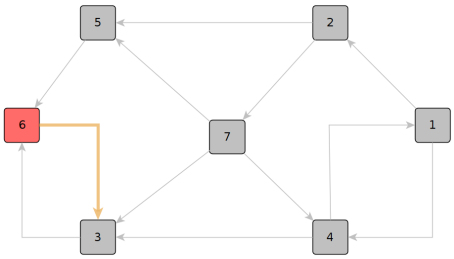



**(b)** Matrix representation of step 3

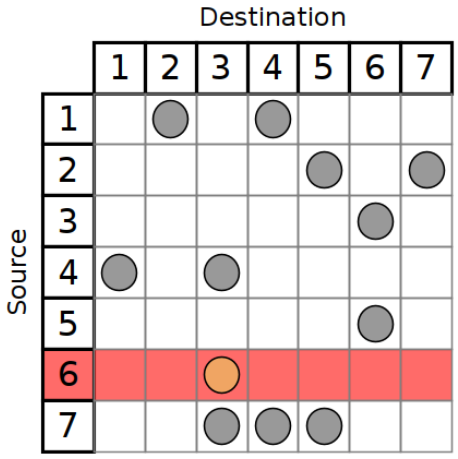



**(c)** Arrays after step 3

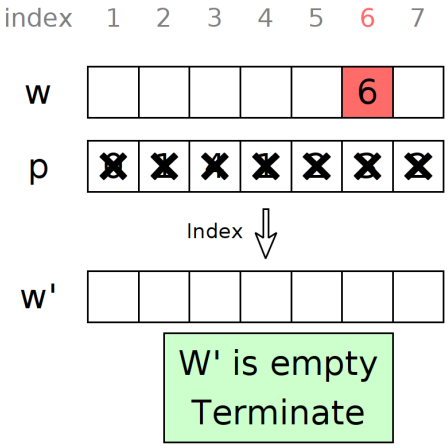
**Figure B.4:** Step 3




**(a)** Graph representation of step 4




**(b)** Matrix representation of step 4




**(c)** Arrays in final state

**Figure B.5:** Any shortest path step 4 / final state

# Appendix C

# Literature study inclusion

| # | Title | Year | Included | I1 | I2 | I3 | I4 | I5 | I6 | I7 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Graph Pattern Matching in GQL and SQL/PGQ | 2021 | x | | | x | x | x | x | x |
| 2 | Query, Analysis, and Benchmarking Techniques for Evolving Property Graphs of Software Systems | 2019 | x | x | | x | x | x | x | x |
| 3 | Cypher: An Evolving Query Language for Property Graphs | 2018 | x | x | | x | x | x | x | x |
| 4 | G-CORE A Core for Future Graph Query Languages | 2018 | x | x | | x | x | x | x | x |
| 5 | Knowledge Graphs | 2021 | x | | | x | | x | x | x |
| 6 | An early look at the LDBC Social Network Benchmark's Business Intelligence workload | 2018 | x | | | x | x | x | x | x |
| 7 | The property graph database model | 2018 | x | | | | x | x | x | x |
| 8 | Two for one: querying property graph databases using SPARQL via gremlinator | 2018 | x | | | x | x | x | x | x |
| 9 | MilleniumDB: A Persistent, Open-Source, Graph Database | 2021 | x | x | | x | | x | x | x |
| 10 | GrainDB: A Relational-core Graph-Relational DBMS | 2022 | x | x | | x | | x | x | x |
| 11 | Efficient Batched Graph Analytics Through Algorithmic Transformation | 2017 | x | | x | | | x | x | x |
| 12 | The more the Merrier: Efficient Multi-Source Graph Traversal | 2014 | x | | x | | | x | x | x |
| 13 | Demystifying Graph Databases: Analysis and Taxonomy of Data organization, Systems design and Graph queries | 2019 | x | x | | | x | x | x | x |
| 14 | The case against specialized graph analytics engines | 2015 | x | x | | x | | x | x | x |
| 15 | PGQL: a Property Graph Query Language | 2016 | x | | | x | | x | x | x |
| 16 | The Complete Story of Joins (inHyPer) | 2017 | | x | | | | x | x | x |
| 17 | The ubiquity of large graphs and surprising challenges of graph processing: extended survey | 2020 | x | | | | x | x | x | x |
| 18 | Trinity Graph Engine and its Applications. | 2017 | x | x | | | | x | x | x |
| 19 | SQLGraph: An Efficient Relational-Based Property Graph Store | 2015 | x | x | | | x | x | x | x |
| 20 | Building an efficient RDF store over a relational database | 2013 | x | x | | | x | x | x | x |
| 21 | The RDF-3X engine for scalable management of RDF data | 2010 | x | x | | | | x | x | x |
| 22 | GraphGen: Adaptive Graph Processing using Relational Databases | 2017 | x | x | | x | | x | x | x |
| 23 | IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2 | 2020 | x | x | | | | x | x | x |
| 24 | Graph Processing in RDBMSs | 2017 | x | x | | | | x | x | x |
| 26 | Declarative and distributed graph analytics with GRADOOP | 2018 | x | x | | | | x | x | x |
| 27 | Two for one: querying property graph databases using SPARQL via gremlinator | 2018 | x | x | | | x | x | x | x |
| 28 | The case against specialized graph analytics engines | 2015 | x | x | x | | x | x | x | x |
| 29 | Cytosm: Declarative Property Graph Queries Without Data Migration | 2017 | x | x | | x | x | x | x | x |
| 30 | Foundations of Modern Query Languages for Graph Databases | 2017 | x | x | | x | x | x | x | x |
| 31 | Columnar Storage and List-based Processing for Graph Database Management Systems | 2021 | | x | | | | x | x | x |
| 32 | Making RDBMSs Efficient on Graph Workloads Through Predefined Joins | 2021 | x | x | | x | | x | x | x |
| 33 | Bank Fraud Detection using Community Detection Algorithm | 2020 | x | | | x | | x | x | x |
| 34 | Certified Graph View Maintenance with Regular Datalog | 2018 | | | | | | x | x | x |
| 35 | Modeling of Emergency Evacuation in Building Fire | 2020 | | | x | | | x | x | x |
| 36 | Aggregation Support for Modern Graph Analytics in TigerGraph | 2020 | x | | | | | x | x | x |
| 37 | Finding Emergent Patterns of Behaviors in Dynamic Heterogeneous Social Networks | 2019 | | | | x | | x | x | x |
| 38 | Integration of Relational and Graph Databases Functionally | 2019 | | | | | x | x | x | x |
| 39 | Fast dual simulation processing of graph database queries | 2019 | x | | | x | | x | x | x |
| 40 | Efficient graph pattern matching framework for network-based in-vehicle fault detection | 2018 | | | | x | | x | x | x |
| 41 | Minimization of tree patterns | 2018 | | | | | | x | x | x |
| 42 | In-Depth Benchmarking of Graph Database Systems with the Linked Data Benchmark Council (LDBC) Social Network Benchmark (SNB) | 2019 | x | x | | | x | x | x | x |
| 43 | TigerGraph: A Native MPP Graph Database | 2019 | x | x | | x | x | x | x | x |