Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Integrating SQL/PGQ into DuckDB

Author: Daniël ten Wolde (2619049)

1st supervisor: Prof. dr. 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 15, 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) to formulate graph pattern matching and path-finding operations using a concise syntax. These features allow the extension of existing, mature relational database management systems (RDBMSs) with efficient execution techniques for graph queries, including graph pattern matching and path-finding. However, as of 2022, SQL/PGQ is not yet implemented in any RDBMS, lacking both a reference implementation and an optimized high-performance implementation. Moreover, research on integrating path-finding algorithms into RDBMSs has been limited.

In this thesis, we investigate the feasibility of integrating path-finding algorithms in the open-source RDBMS DuckDB: unweighted shortest path, cheapest path, and any shortest path. Using a lightweight extension approach that relies on scalar 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), a state-of-the-art benchmark suite for testing graph functionalities and performance of DBMSs. The experiments show that the implemented algorithms scale close to linearly and are able to handle the largest graphs currently available in the LDBC SNB. The results demonstrate that DuckDB is a suitable 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 7

2.1 Property Graph Data Model . . . 7

2.2 SQL/Property Graph Queries . . . 8

2.3 DuckDB . . . 11

2.3.1 DuckDB Query Execution Pipeline . . . 12

2.3.2 Vectorised Execution Engine . . . 13

2.3.3 Join Operator . . . 13

2.3.4 Scalar User-Defined Functions . . . 13

2.4 Current State of SQL/PGQ in DuckDB . . . 14

2.4.1 Overview . . . 14

2.4.2 Compressed Sparse Row . . . 16

2.5 Single Instruction, Multiple Data Execution Model . . . 20

2.6 Cache Latency . . . 23

3 Related Work 25

3.1 Resource Description Framework . . . 25

3.2 Graph Query Languages . . . 26

3.2.1 Survey on Graph Query Languages . . . 26

3.2.2 Cypher . . . 27

3.2.3 G-CORE . . . 28
3.2.4 Property Graph Query Language (PGQL) . . . 29
3.2.5 GSQL . . . 29
3.2.6 SPARQL . . . 29
3.2.7 Gremlin . . . 30
3.2.8 Graph Query Language . . . 30
3.3 Mapping Strategies . . . 31
3.3.1 Mapping from Graph to Relational Queries . . . 31
3.3.2 Mapping from RDF to PG . . . 31
3.4 Graph Traversal Algorithms . . . 32
3.4.1 Unweighted Shortest Path . . . 33
3.4.2 Cheapest Path (Weighted Shortest Path) . . . 35
3.5 Survey on Graph Database Management Systems . . . 39
3.6 Data Processing Systems Supporting Graph Processing Workloads . . . 40
3.6.1 Neo4j . . . 40
3.6.2 Umbra . . . 40
3.6.3 TigerGraph . . . 41
3.6.4 GRADOOP . . . 41
3.6.5 The Case Against Specialised Graph Analytics Engines . . . 42
3.6.6 GRainDB . . . 42
3.6.7 MillenniumDB . . . 43

4 Design & Implementation **45**
4.1 Overview . . . 45
4.2 Shortest Path . . . 46
4.3 Any Shortest Path . . . 49
4.4 Weighted Compressed Sparse Row . . . 53
4.5 Cheapest Path . . . 54
4.5.1 Batched Bellman-Ford implementation . . . 54
4.5.2 Vectorising Batched Bellman-Ford . . . 57
4.6 Shared Hash Join . . . 58

5 Evaluation **63**
5.1 Experimental Setup . . . 63
5.1.1 Environments . . . 63
5.1.2 LDBC Social Network Benchmark . . . 64
5.2 Shortest Path Length . . . 67
5.3 Cheapest Path Length . . . 71
5.3.1 Interactive Query 13 . . . 71
5.3.2 Comparison Between Shortest and Cheapest Path Length . . . 71
5.3.3 BI Query 19a . . . 75
5.3.4 BI Query 20 . . . 77
5.4 Vectorised Batched Bellman-Ford . . . 79
5.5 Shared Hash Join . . . 82

6 Future Work **85**

7 Conclusion **89**

References **93**

A SQL vs. SQL/PGQ Queries **109**

B Any Shortest Path Example **111**

C Summary of Literature Study **113**

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 7
2.2 Social network graph of Person nodes and the Universities where they study. 9
2.3 Query execution pipeline of DuckDB 12
2.4 Overview of SQL/PGQ execution in DuckDB by Singh et al. 16
2.5 Unweighted directed graph and the CSR representation 17
2.6 Vertex and edge tables related to the graph shown in Figure 2.5a 18
2.7 The CSR vertex array before and after executing the `create_csr_vertex` function 18
2.8 The CSR vertex array and edge array after executing `create_csr_edge` . . 19
2.9 Scalar vs. vectorised instructions 20
2.10 Comparison between non-SIMD-ised and SIMD-ised compiler outputs . . . 21
2.11 x86-64 gcc 9.4 output with SIMD instructions. 23

4.1 Schematic overview of key components for efficient path-finding operators . 46
4.2 MS-BFS initial state 50
4.3 MS-BFS step 1 50
4.4 MS-BFS step 2 50
4.5 MS-BFS step 3 (final) 51
4.6 Any shortest path initial state 51
4.7 Any shortest path step 1 52
4.8 Any shortest path step 2 52
4.9 Weighted directed graph and the CSR representation 53
4.10 Example of a duplicate sink state in the physical plan 60

5.1 LDBC Interactive Query 13 (1) . . . 64
5.2 LDBC BI Query 19 (2) . . . 65
5.3 LDBC BI Query 20 (3) . . . 66
5.4 Average execution time per scale factor for shortest path Interactive query 13 67
5.5 Average execution time per source-destination pair per scale factor for shortest path Interactive query 13 . . . 68
5.6 Relative time spent per phase for Interactive query 13 shortest path . . . 69
5.7 CSR creation time for all scale factors . . . 69
5.8 CSR creation using a single thread . . . 70
5.9 Total execution time per scale factor for the cheapest path variant of Interactive query 13 . . . 71
5.10 Average execution time per source-destination pair per scale factor for the cheapest path variant of Interactive query 13 . . . 72
5.11 Relative time spent per phase for Interactive query 13 cheapest path . . . 72
5.12 Difference in performance between shortest and cheapest path for query 13 . 73
5.13 Average execution time per scale factor for cheapest path BI query 19a . . . 76
5.14 Average execution time per source-destination pair per scale factor for cheapest path BI query 19a . . . 76
5.15 Relative time spent per phase for cheapest path BI query 19a . . . 77
5.16 Average execution time per scale factor for cheapest path BI query 20 . . . 78
5.17 Average execution time per source-destination pair per scale factor for cheapest path BI query 20 . . . 79
5.18 Relative time spent per phase for cheapest path BI query 20 . . . 80
5.19 Performance of scalar vs. auto-vectorised implementations of batched Bellman-Ford . . . 81
5.20 Performance comparison of the shared hash join optimisation . . . 83

B.1 Any shortest path initial state . . . 111
B.2 Any shortest path step 1 . . . 111
B.3 Any shortest path step 2 . . . 112
B.4 Any shortest path step 3 . . . 112
B.5 Any shortest path step 4 / final state . . . 112

List of Tables

2.1 Event with corresponding latency, also scaled to 1 second 23

3.1 Variatons of a graph traversal algorithms for various scenarios 33

5.1 Graph size and number of source-destination pairs per scale factor for Interactive query 13 67

5.2 Graph size and number of source-destination pairs per scale factor for BI query 19a 75

5.3 Graph size and number of source-destination pairs per scale factor for BI query 20 78

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* (4). A comprehensive survey of Sahu et al. (5) 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) (6).

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 (6, 7), 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 (8).

In response to the limited capabilities, a plethora of GDBMSs arose in the last fifteen years, each having its graph query language (9). Examples of systems and their query languages are TigerGraph with GSQL (10), Neo4j with Cypher (11), and Oracle Labs PGX with PGQL (12). 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 currently being developed by the *ISO/IEC JTC1 SC32 WG3 Database Languages* working group in a liaison with the Linked Data Benchmark Council (13) (LDBC) which is founded and led by the CWI Database Architectures group. SQL/Property Graph Queries has been partially based on graph query language research done by LDBC (14) and is scheduled to release in June of 2023 as part of the upcoming SQL:2023 standard (15).

SQL/PGQ will make working with graph-based data in RDBMSs considerably easier (15). In SQL/PGQ, a graph can be defined in terms of tables (16) and queries can contain special syntax for path-finding and graph pattern matching.

For querying graph data, two functionalities are deemed most important: *graph pattern matching* and *path-finding* (17). 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 (18). In particular, path-finding requires using recursive queries. An example can be seen in Listing 1.1, which shows a path-finding query using SQL:1999 syntax 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' (19). Listing 1.1 makes use of `WITH RECURSIVE` that starts with an initialisation query. This is unioned (using `UNION ALL`) with a recursive query that joins the paths. In Listing 1.1, paths are represented as a list of vertices, shown in red. In contrast, the same query in SQL/PGQ is shown in Listing 1.2. The `WITH RECURSIVE` can be replaced with the `+` operator that represents the Kleene plus, indicating that the pattern can occur 1 or more time(s).

```
1  WITH RECURSIVE paths(startNode, endNode, path) AS (
2      SELECT -- Initialisation
3          p1id AS startNode,
4          p2id AS endNode,
5          [p1id, p2id] AS path
6        FROM follows
7      UNION ALL
8      SELECT -- Recursion
9          paths.startNode AS startNode,
10         p2id AS endNode,
11         array_append(path, p2id) AS path
12       FROM paths
13       JOIN follows ON paths.endNode = p1id
14       WHERE p2id != ALL(paths.path)
```

```
15      )
16  SELECT count(p2.id) AS cp2
17  FROM person p1
18  JOIN paths     ON paths.startNode = p1.id
19  JOIN person p2 ON p2.id = paths.endNode
20  JOIN city      ON city.id = p2.livesIn AND city.name = 'Delft'
21  WHERE p1.name = 'Daniel';
```

Listing 1.1: SQL:1999 query using `WITH RECURSIVE`

```
1  SELECT count(gt.id) AS cp2
2  FROM GRAPH_TABLE (socialNetwork,
3    MATCH
4     (p1:person WHERE name = 'Daniel')-[:follows]->+
5     (p2:person)-[:livesIn]->(c:city WHERE name = 'Delft')
6    COLUMNS (p2.id)
7  ) gt
```

Listing 1.2: SQL/PGQ query equivalent to Listing 1.1 using the Kleene plus operator (+)

The `RECURSIVE` statement was added in SQL:1999 (20), and is supported by popular RDBMSs such as PostgreSQL, MySQL, and SQLite. However, as shown by Michels and Witkowski (8), 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.

SQL/PGQ is limited to read-only queries, making it impossible to modify the graph through the graph tables. Therefore, the same workgroup is also working on Graph Query Language (GQL) (21), 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 (22). In future versions, it will also be possible to return graphs as a result of a query in GQL, a feature which is not supported by any graph query languages to date (14).

In this thesis, we integrate parts of the SQL/PGQ standard in DuckDB (23). DuckDB is an open-source in-process SQL OLAP database management system originating from CWI (24) that uses a vectorised query execution (25). With the new SQL/PGQ standard releasing soon, work on integrating SQL/PGQ into DuckDB has already started by Singh

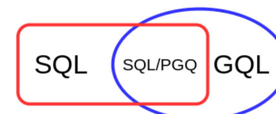


Figure 1.1: SQL/PGQ in relation to SQL and GQL

et al. (26). However, SQL/PGQ was not been fully implemented at the start of this project. In particular, certain path-finding functionalities 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, and another in which edges have a numerical (but typically positive) weight.

In the unweighted case, only reachability had been implemented. In this project, we added algorithms for returning the shortest path length (in number of hops), as well as returning the shortest path (as an array of vertices). For the weighted case, we needed to add an algorithm for finding the cheapest paths, where the path contains the minimum sum of weights and return this cost. Therefore, the goal is to further complete the SQL/PGQ integration in DuckDB. Finally, we will identify and implement optimisations primarily related to typical SQL/PGQ queries expected to be ran in DuckDB.

The implementation will be done while respecting the design principles followed by Singh et al. (26). This included making sure that the integration is non-intrusive and can be largely limited to a DuckDB extension module; yet achieves high efficiency, leveraging the parallelism and vectorised query processing of 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, the upcoming SQL/PGQ standard, DuckDB and the current state of the SQL/PGQ implementation in DuckDB. Chapter 3 shows related work, such as existing graph query languages and various path-finding algorithms. Chapter 4

discusses the most important design and implementation details, and describes how the various path-finding algorithms and optimisations have been implemented. Chapter 5 shows the results of the experiments conducted and provides an in-depth discussion on the results obtained. Finally, Chapter 6 and Chapter 7 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. It models persons residing in a city, who joined the social network, formed friendships (*knows* relationships), studied at some university and worked at some company.

2.1 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, referred to as relationships, represent the connections (relations) between objects. The simplest form of a graph is the *simple directed graph model* (9). In this model, the graph consists of a set of vertices and a set of edges. Each edge has a source vertex and a destination vertex. In the case of an

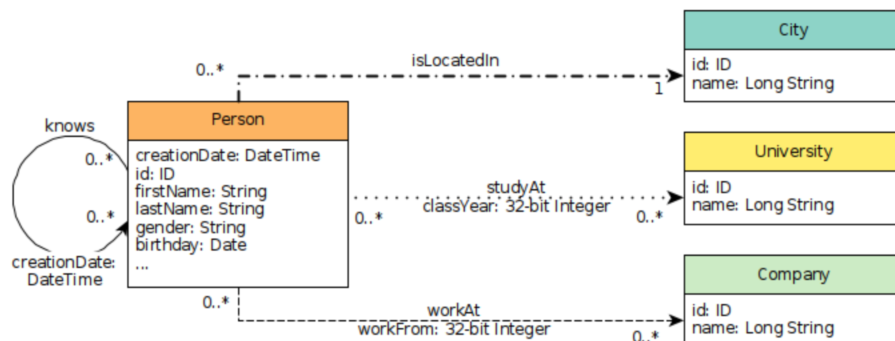


Figure 2.1: Partial LDBC Social Network Benchmark schema depicted using a UML-like notation

undirected graph, both vertices within an edge act as the source and destination vertex. Graphs can either be weighted or unweighted. In weighted graphs, an edge between two vertices 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 (9).

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. There is the *knows* label for which the source vertex is of label *Person*, and the destination vertex is also of label *Person*. Additionally, there is *studyAt* which starts from a *Person* and points to a *University*.

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 (27), TigerGraph (10), and Oracle Labs PGX (28).

2.2 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 (8). The two most crucial graph querying functionalities are graph pattern matching and path-finding as described by Angles et al. (17). These functionalities become more accessible with the addition of SQL/PGQ, and queries involving these can be more easily expressed (8, 18).

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 (29). A graph can be defined using the SQL statement (30) found in Listing 2.1.

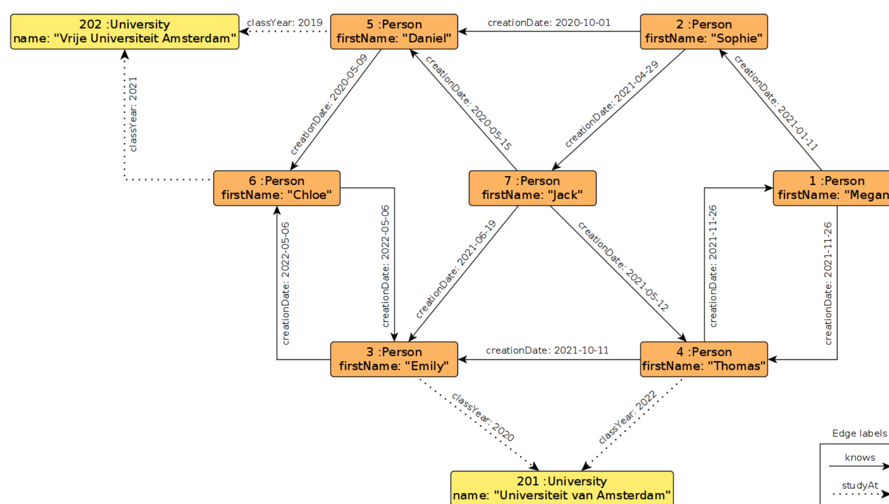


Figure 2.2: Social network graph of Person nodes and the Universities where they study.

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

Listing 2.1: Clause for creating a graph in SQL/PGQ

For example, if we wish to create the schema of the graph of Figure 2.2, symbolising a group of friends, some of whom study at a university, we would use the following query:

```
CREATE PROPERTY GRAPH social_network
VERTEX TABLES (
    Person PROPERTIES ( personId, firstName ),
    University PROPERTIES ( universityId, name )
)
EDGE TABLES (
    knows SOURCE Person DESTINATION Person PROPERTIES ( creationDate ),
    studyAt SOURCE Person DESTINATION University PROPERTIES ( classYear )
)
```

Listing 2.2: Creating the schema of the example social network graph of Figure 2.2 in SQL/PGQ

To match a pattern to this graph in SQL/PGQ, the `MATCH` syntax can be used (22), as can be seen in Listing 2.3. For example, Listing 2.3 selects the personid and first name of persons from the graph in Figure 2.2 whose name is equal to 'Daniel'.

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

Listing 2.3: Pattern matching all vertices 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 2.3, we use the `()` notation to denote a vertex. Denoting an edge can be done using `[]`.

To indicate that an edge is pointing from source to destination we write `(source)-[` *spec* `]->(destination)` (22). 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 3.2.2.

A more complex graph pattern involving both vertices and undirected edges is shown below:

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

Listing 2.4: Graph pattern matching using vertices and edges of a subgraph

This statement extracts all patterns that match vertex *a* being a Person with the name "Jack", who knows a person studying at a university. Within every vertex or edge, we can filter the possibilities by adding a `WHERE` statement, as seen in Listing 2.4.

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 (22). 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 2.5: 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 (8). 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:

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

Listing 2.6: MATCH clause defining a path of arbitrarily many knows edges

An example of a Regular Path Query (RPQ, presented in Section 3.2.1) possible in SQL/PGQ can be seen in Listing 2.7.

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

Listing 2.7: MATCH clause defining a graph pattern with a repeating sequence of edges

Queries in SQL/PGQ can also perform path-finding operations. One can specify that *any* shortest path matching the graph pattern should be returned using the `ANY SHORTEST` keyword. The `ANY` feature has the known disadvantage of being non-deterministic as there can be many shortest paths, all of the same length, and the language feature serves to return only one path as a query result. As a query result, `ANY` allows for a single arbitrary shortest path to be returned.

Another option is to return *ALL* shortest paths, which is deterministic. It is possible to return the shortest k paths, which is also a non-deterministic operation in case there are multiple paths with the same length as the kth result. Finally, in SQL/PGQ, it is possible to return the vertices or edges found on a path corresponding to a query.

Performing path-finding operations on weighted graphs (e.g. finding cheapest paths) is marked as a *language opportunity* in the first stable release of SQL/PGQ (22). Therefore, while not yet officially part of the language, this feature is likely to be included in later versions of SQL/PGQ. In this project, we have already sought to implement this feature.

2.3 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 (23). Like SQLite, DuckDB is an in-process system, though SQLite is specialised in transactional workloads.

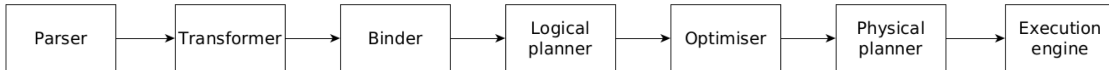


Figure 2.3: Query execution pipeline of DuckDB

DuckDB consists of several components: parser, transformer, logical planner, optimiser, physical planner, and execution engine, see Figure 2.3. The system can be accessed through a C/C++ API and has an SQLite compatibility layer.

2.3.1 DuckDB Query Execution Pipeline

The SQL parser is based on the PostgreSQL SQL parser (23). 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 relational algebra operators from the retrieved parse tree.

The optimiser will then create an optimised logical plan 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 (31). 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. 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). 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. Sink operators such as sorting, building a hash table, and aggregation must access all the data before proceeding. All other operators do not need to materialise all data before proceeding. In the case of such binary operators, there are 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.

Pipelines can run on multiple cores in parallel. Parallelisation is done using the morsel-driven method (32). DuckDB will schedule work for one "morsel" on a thread for every $120 \cdot 1024 = 122\,880$ tuples from the pipeline source for each core at a time. This threshold can be adjusted using the pragma `verify_parallelism`, which will schedule a morsel for

every vector of 1024 tuples on a thread, though this is only used for debugging purposes. With a pragma, the internal properties of the database engine can be adjusted.

2.3.2 Vectorised Execution Engine

The execution engine of DuckDB is vectorised (23). The vectorised query execution engine was pioneered by VectorWise (25, 33). Vectorisation uses pull-based iterations where every operator fetches the next set of tuples using a *next* operation. Instead of fetching a single tuple, done with Volcano-like tuple-at-a-time execution (34), it fetches a larger number of tuples (35). DuckDB has a standard vector size of 1024 tuples, which is also said to be the optimal vector size by Boncz et al. (25). Boncz et al. also found that vector sizes between 128 and 8k work well. When vectors no longer fit inside the cache, performance starts to get worse (25). 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 (25). The size of vectors can be adjusted in DuckDB, though this is typically only done for debugging purposes.

2.3.3 Join Operator

The join operation is a vital relation operation that performs a join between two tables. For every join operator, a hash table needs to be built on which the join can be performed[1]. The operator must wait for the entire hash table to be built before proceeding with the join operation. 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.

2.3.4 Scalar User-Defined Functions

In DuckDB it is possible to register scalar user-defined Functions (UDFs) in extension modules that can be used in SQL expressions. These scalar UDFs are as fast as the built-

[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 (36).

in functions of DuckDB due to the vectorised query processing, whereby DuckDB handles the parallelisation of the UDF since the expression evaluation is part of the operator evaluation, which in turn is part of the pipeline evaluation. We will use scalar UDFs for the path-finding and CSR creation functions to create a lightweight implementation. 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.

DuckDB allows for the creation of extension modules. These extension modules are separate from the mainline, core DuckDB and are not always included by default in every distribution. They can be remotely installed by the user whenever required.

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.

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. The client context is also shared between parallel threads, which is useful when creating the CSR explained in Section 2.4.2.

2.4 Current State of SQL/PGQ in DuckDB

2.4.1 Overview

This thesis will use the work done by Singh et al. at CWI (26). 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 visual graph style query syntax introduced with SQL/PGQ as shown in Section 2.2. New keywords like `GRAPH`, `LABEL` and `PROPERTIES` were added to the parser to allow correct parsing of SQL/PGQ queries. A parser is not extensible; therefore, the changes made to the parser are part of the core of DuckDB, unlike the scalar UDFs, which are implemented as part of an extension module. The binder translates the newly introduced `MATCH` to `JOIN` operations, thus, the SQL/PGQ queries are transformed into SQL:1999 queries. Listing 2.8 contains

```
1  SELECT c1id, c2id, c3id
2  FROM GRAPH_TABLE (aml,
3    MATCH
4      (c1:customer)-[t1:transfers]->(c2:customer)-[t2:transfers]->
5      (c3:customer)-[t3:transfers]->(c1)
6    COLUMNS (c1.cid AS c1id, c2.cid AS c2id, c3.cid AS c3id)
7  ) gt;
```

Listing 2.8: SQL/PGQ query looking for 3-edge cycles between `customers` along `transfers` edges

```
1  SELECT c1id, c2id, c3id
2  FROM (
3    SELECT c1.cid as c1id, c2.cid as c2id, c3.cid as c3id
4    FROM
5      customer c1, transfers t1, customer c2, transfers t2,
6      customer c3, transfers t3
7    WHERE c1.cid = t1.from_id
8      AND c2.cid = t1.to_id
9      AND c2.cid = t2.from_id
10     AND c3.cid = t2.to_id
11     AND t3.from_id = c3.cid
12     AND t3.to_id = c1.cid
13 );
```

Listing 2.9: SQL:1999-compliant implementation of the query in Listing 2.8

an example SQL/PGQ query with a `MATCH` statement followed by a graph pattern of vertices and edges. This is translated in the binder such that every *vertex to edge* (`()-[]`) and *edge to vertex* (`[]-()`) is a `JOIN` operation in the resulting SQL:1999 query shown in Listing 2.9. Singh et al. have also implemented the functionality to compute the reachability between two vertices in a graph. Computing the reachability was implemented as a scalar UDF using the Multi-Source Breadth-First Search (MS-BFS) algorithm by Then et al. (37), a more in-depth explanation of the algorithm is provided in Section 4.2.

Figure 2.4 shows a schematic overview of the elements worked on by Singh et al. Step ① shows the `CREATE PROPERTY GRAPH` which defines the property graphs over the relational tables previously created with SQL. This step only registers the view and does not do any data processing yet. Step ② is the path-finding query which leads to the on-the-fly creation of a Compressed Sparse Row (CSR) data structure through steps ③–⑤, and to the final query results through ⑥ and ⑦. Step ③ labels the vertices with numeric ids from a dense contiguous range to represent the graph with good memory locality and neighbourhood

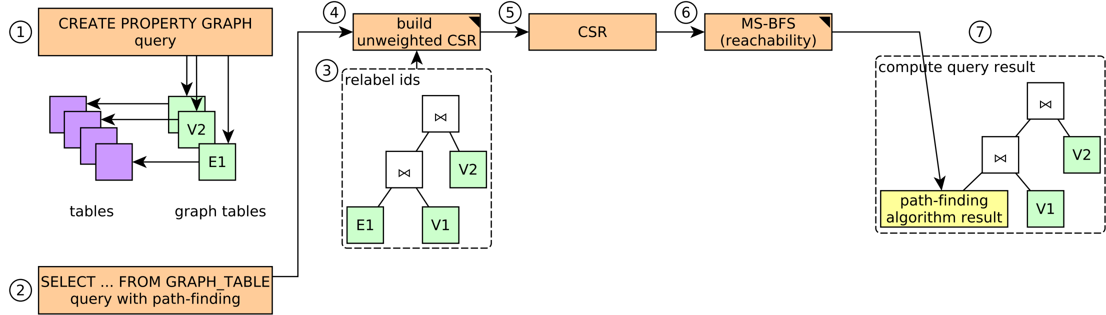


Figure 2.4: Overview of SQL/PGQ execution in DuckDB by Singh et al.

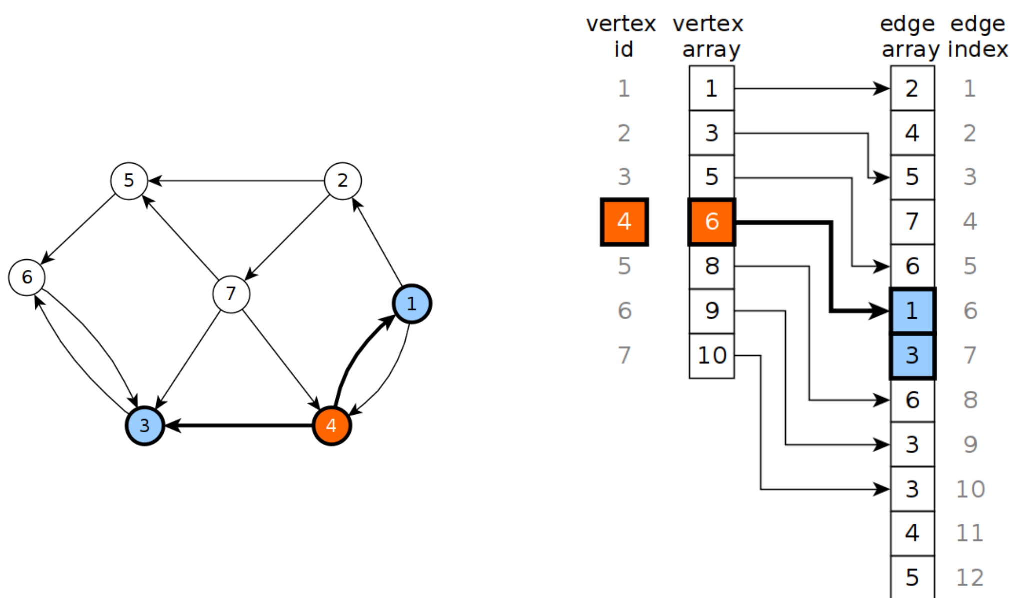
lookups. Step ④ builds a CSR data structure, resulting in step ⑤. Step ⑥ then uses the CSR to compute the reachability between vertices using MS-BFS.

2.4.2 Compressed Sparse Row

An intuitive way to represent graphs in programs is using a pointer between objects (vertices). In this case, every node contains a list of pointers to all nearby nodes (38). However, this becomes inefficient in the case of multiple traversals. The edges in a graph may be irregular and unstructured, resulting in the data access patterns not having good memory locality (39), leading to random memory accesses.

The Compressed Sparse Row (CSR) data structure can be used to create a compact graph data structure with a good locality. The locality is vital for the shortest and cheapest path algorithms, as they are not computationally heavy but require intensive memory access (37). Therefore, it is essential to reduce the number of cache misses.

Figure 2.5 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.



(a) Unweighted directed graph

(b) CSR representation of unweighted directed graph

Figure 2.5: Unweighted directed graph and the CSR representation

We chose to create the CSR on-the-fly just before the shortest or cheapest path function is executed. The on the fly creation was chosen for two reasons. First, CSR creation will likely not take much time relative to the shortest and cheapest path functions. Second, when creating the CSR on-the-fly, we do not need to worry about maintaining the CSR whenever the graph is updated. Updating a CSR data structure is a difficult operation (40).

We use the row identifiers of the vertex table to represent vertices in the CSR data structure. The ids of the vertices in the vertex table need not be in order for the CSR to work. In DuckDB, the CSR is created in two separate scalar UDFs.

The first scalar UDF, `create_csr_vertex`, is concerned with initialising the vertex array. The vertex array consists of 64-bit integers. The parameters are an ID given by the user, the total number of vertices and a dense id of a vertex and the number of outgoing edges of that vertex. At first, the vertex array will only hold the number of outgoing edges per vertex, which will later be used to compute the offsets for the edge array by computing a running sum. Returned is the number of outgoing edges per vertex, the same as the last parameter.

The second scalar UDF, `create_csr_edge`, is related to initialising the edge array, which can only be done after the vertex array has been initialised. It modifies the offsets in the vertex array using a running sum and fills the edge array with the destination vertices. In

Vertex table

ID
1
2
3
4
5
6
7

Edge table

Source	Destination
1	2
1	4
2	5
2	7
3	6
4	1
4	3
5	6
6	3
7	3
7	4
7	5

Figure 2.6: Vertex and edge tables related to the graph shown in Figure 2.5a

Source	# of outgoing edges
1	2
2	2
3	1
4	2
5	1
6	1
7	3

(a) Input of the function

Index	CSR vertex array
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0

(b) Before executing the function

Index	CSR vertex array
1	0
2	2
3	2
4	1
5	2
6	1
7	1
8	3

(c) After executing the function

Figure 2.7: The CSR vertex array before and after executing the `create_csr_vertex` function

order to determine where the edge goes in the edge array, it looks up the offset value of the vertex before the source vertex and increments the offset by one using an atomic. The parameters are the CSR ID, the number of vertices, the total number of edges and the vertex table dense ids of all source and destination vertices found in the edge table.

At the time of designing these functions, DuckDB did not allow for sorting to be done in parallel[1]. This was one of the reasons to use an approach that avoids using `ORDER BY` was chosen.

Upon creation, a CSR is stored in the client context. After the shortest or cheapest path functions have been finalised, the CSR will be deleted.

Source	Destination
1	2
1	4
2	5
2	7
3	6
4	1
4	3
5	6
6	3
7	3
7	4
7	5

(a) The dense IDs of all the source and destination vertices used in `create_csr_edge`

Index	CSR vertex array
1	1
2	3
3	5
4	6
5	8
6	9
7	10
8	13

CSR edge array	Index
2	1
4	2
5	3
7	4
6	5
1	6
3	7
6	8
3	9
3	10
4	11
5	12

(b) The CSR vertex array and edge array after executing `create_csr_edge`

Figure 2.8: The CSR vertex array and edge array after executing `create_csr_edge`

Following is an example of the various steps involved in creating the CSR data structure shown in Figure 2.5[2]. The tables used as input are shown in Figure 2.6. Figure 2.7 shows the CSR vertex array before and after executing the `create_csr_vertex` function. In Figure 2.7b, the initial state is shown where the CSR vertex array contains all zero values. The length of the array is $|V|+1$ to add padding for the rolling sum used later. The number of outgoing edges for a vertex is inserted in the vertex array at the position `Source ID + 1`. Figure 2.8 shows parts of the input to the `create_csr_edge` function (the number of vertices and edges have been omitted) and the CSR vertex array and edge array after

[1] The latest DuckDB version (0.4.0), allows sorting to be done in parallel (41), however in this thesis, work is performed on an older DuckDB version (0.2.2) which did not yet have this feature

[2] The arrays in the example make use of 1-based indexing

execution the function. We observe that the vertex array now contains the offset for the first outgoing edge for every vertex. The last value in the vertex array, 13, is $|E|+1$ and signifies there are no more edges following in the edge array.

2.5 Single Instruction, Multiple Data Execution Model

In order to increase the CPU efficiency, we can use 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 2.9. 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.

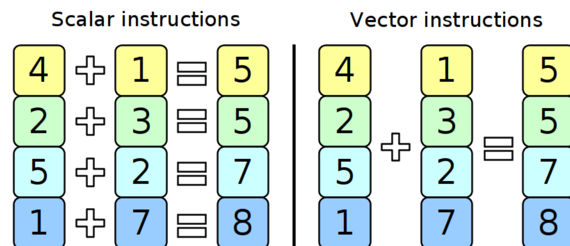


Figure 2.9: Scalar vs. vectorised instructions

An example can be observed in Figure 2.10a[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 2.10b. The instruction `movzx` copies the content of the source (`BYTE PTR [rsp+16+rax]` in this case, to the destination (`edx`). Then 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

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

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) x86-64 GCC 9.4 output without SIMD instructions

```
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) x86-64 GCC 9.4 output with SIMD instructions

Figure 2.10: Comparison between non-SIMD-ised and SIMD-ised compiler outputs

On the other hand, Figure 2.10c 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. In total, only 8 instructions are needed.

Various versions of SIMD instructions exist, with the oldest being the SSE instruction set introduced initially with the Intel Pentium III in 1999 (42). These came with the XMM registers seen in the instructions in Figure 2.10c. The XMM registers are 128-bits wide.

SSE2 was introduced with the Intel Pentium 4 in 2000 (43). It added the support for the double-precision data type, allowing operations to be performed on the full range of data types, from 8-bit integer to 64-bit float. Nowadays, according to the Steam hardware survey, 100% of CPUs in consumer PCs support SSE2 (44)[1].

The Intel Advanced Vector Extensions (AVX) was introduced in 2008 and was first supported by the Sandy Bridge processor (45). It introduced 256-bit wide YMM registers,

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

with the lowest 128 bits identical to the XMM registers (46, Chapter 13). In 2013 AVX-512 was introduced with ZMM registers, which are 512 bits wide. The Steam hardware survey shows that AVX and AVX2 are supported by roughly 90% of all CPUs observed, while AVX-512 is supported by less than 10%.

ARM introduced a small set of SIMD instructions with ARMv6 in October of 2001 (47). ARMv6 introduces special 32-bit registers on which 8-bit or 16-bit values could be operated. Thus certain operations could be executed twice or four times faster. The implementation of the Advanced SIMD instruction set for ARM is called *NEON* and was introduced with the ARMv7 architecture (48). It added support to operate on more data types, such as 64-bit double. NEON consists of 32 64-bit registers that can be used for Advanced SIMD. If needed, these 32 registers can also be viewed as 16 128-bit registers.

SIMD instructions can be generated either implicitly or explicitly. Implicit means that the compiler generates SIMD instructions based on code patterns; an example of this can be seen in Figure 2.10a. However, if the complexity of the code increases, it could be that the compiler cannot generate SIMD-ised instructions automatically. 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.

A disadvantage of using explicit SIMD instructions is a loss of portability across platforms. Therefore, it is not the preference of DuckDB to use explicit SIMD instructions.

An example of explicit SIMD code can be seen in Figure 2.11. Like in Figure 2.10a, there are two 16-length 8-bit arrays A and B, occupying 128 bits. The first two lines of Figure 2.11a are concerned with loading the arrays in the special XMM registers. `_mm_load_si128` corresponds with the `movedqa xmm` instructions seen in Figure 2.11b. `_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 2.11b.

Knowing which specific instructions can be used is essential for the optimal use of SIMD instructions. 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 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 multiple versions of the same functions compiled for different targets, in one library, at compile time. Additionally, it is necessary to use the

```
__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) x86 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) x86-64 gcc 9.4 output using SIMD for adding two arrays

Figure 2.11: x86-64 gcc 9.4 output with SIMD instructions.

open-source C library `cpu_features` (50), which can retrieve CPU features at run-time. The process of selecting the correct function when the program is started is called runtime dispatch. Vectorised engines can amortise the cost of runtime dispatch by installing the function compiled for the most-appropriate target at database startup.

2.6 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.

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	120.0 ns	6 min

Table 2.1: Event with corresponding latency, also scaled to 1 second

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 slower than fetching data from cache levels, as observed in Table 2.1.

If a cache miss occurs and the CPU needs to fetch data, it fetches a chunk of data around the accessed memory. Such a chunk of data is referred to as a cache line, with the whole chunk being the cache line size. Typically, cache lines are 32, 64, and 128 bytes. The theory is that it is likely that the surrounding data will also be required soon.

Smith shows that reading random values from RAM is about an order of magnitude slower than sequentially reading the data (52). This speedup is due to the utilisation of the cache line. In the experiments, single-threaded randomly accessing data was done at 0.46GB/s, while sequentially accessing this data was done at 11.03GB/s. Another technique used is *sequential prefetching* which is a hardware technique that predicts which blocks of data are currently missing in the cache and prefetches these before a cache miss can occur (53).

3

Related Work

3.1 Resource Description Framework

Aside from the property graph data model described in Section 2.1, another data model is the *Resource Description Framework* (RDF) which is a standard data model used for data interchange on the Web (54). Data is modelled in the form of *triples*, which is composed of a *subject*, *predicate*, and an *object* (54). The vertices within RDF can be of three types: Internationalised Resource Identifiers (IRIs) used to globally identify entities and relations, literals, and blank vertices (55). A vertex attribute is modelled as an extra outgoing edge (being the predicate) to another vertex which is the object stored as a literal. A way of modelling edge attributes is described by Sun et al. (56), 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 (56). Database management systems such as Blazegraph (57) and Amazon Neptune (58) have based their data model on RDF.

Storing RDF data in relational databases comes with several challenges (59). 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 (60). In particular, SPARQL queries result in many more joins, causing challenges for query optimisation, query execution and locality of data access (7, 60).

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, given a single table `triples`, returning the attributes `name`, `birthday`, and `location` requires the execution of the following query:

```sql
SELECT
  triples1.object AS name,
  triples2.object AS birthday,
  triples3.object AS location
FROM triples triples1
JOIN triples triples2
  ON triples2.subject = 1
 AND triples2.predicate = 'birthday'
JOIN triples triples3
  ON triples3.subject = 1
 AND triples3.predicate = 'location'
WHERE triples1.subject = 1
  AND triples1.predicate = 'name';
```

Listing 3.1: Self-joins for collecting the attributes of the subject with identifier `1` (61)

Szárnyas describes several challenges related to self-joins (61). One of the challenges is related to the hash-joins often used by database systems to evaluate join operations. Whenever a join between two tables is performed, the smaller of the two tables is used to build a hash table. The other table is then used to probe the hash table and look for matches. However, with self-joins, both tables are equal in size, negating the advantage of building a hash table on a smaller table. In addition, the resulting table will probably be large when the table is of a many-to-many relation.

3.2 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 (18). To query graph data, two functionalities are deemed most important: *graph pattern matching* and *path-finding* (17). 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.

3.2.1 Survey on Graph Query Languages

Angles et al. (17) provide a survey on the foundational features underlying modern graph query languages. They note three critical ingredients for these query languages: a graph data model, graph patterns, and navigational expressions.

Two graph data models are highlighted by the authors: *edge-labelled graphs* and *property graphs*, discussed in Section 2.1. In the edge-labelled graph data model, labels can be assigned to edges to indicate the relationship between vertices. The edge-labelled graph data model can be seen as the basis for the RDF model described in Section 3.1.

Several query languages have emerged to extract data from instances of graph data models. One of the earliest used to query RDF graphs was SPARQL, see Section 3.2.6 and is standardised by W3C (54). To query property graphs, the authors highlight two query languages: Cypher, discussed in Section 3.2.2, and Gremlin (62), discussed in Section 3.2.7. The languages vary in style, purpose, expressivity, and implementation though they share a core of being able to perform *graph pattern matching* and *graph navigation* operations.

Graph pattern matching results from the pattern over the graph data (17). Basic graph patterns match a graph-structured query against a graph database. These patterns can be augmented using features such as projection, union, optional, and difference. For more details on graph pattern matching, we refer the reader to (17).

Graph pattern queries are of a bounded nature. In the case where we wish to follow a path of arbitrary length, we can make use of navigational queries. These are path queries in combination with basic graph patterns. A fundamental path query is the *path existence*, in which we ask if a directed path between two vertices in a graph exists. In theory, this path can follow edges regardless of their label. However, in practice, one needs to restrict the label type of the edge to limit the search space. Path queries can be specified using regular expressions, such as concatenating two paths, applying a union/disjunction of paths, and applying a path zero or more times. Such queries are referred to as Regular Path Queries (RPQs). For more details, we refer the reader to (17).

3.2.2 Cypher

Francis et al. (11) describe Cypher, a language for querying and updating property graph databases and formalise its read-only core. The language started in Neo4j (63), a property graph database system, but has since also been implemented in other industrial products such as SAP HANA Graph (64) and Redis Graph (65). It has also been used for academic research. For example, Sarma et al. used Cypher and Neo4j to detect bank fraud (66). Another example is Cytosm by Steer et al., which converts Cypher queries into plain SQL queries (67), see Section 3.3. Cypher is based on the LPG data model, just like G-CORE (14), see Section 3.2.3, and PGQL (12), see Section 3.2.4.

According to its authors, queries are structured linearly in Cypher, which allows users to think of query processing as starting from the beginning and progressing linearly to the

end (11). Whereas SQL queries start with a `SELECT` statement listing the attributes that comprise the query result, Cypher queries end with a `RETURN` statement. An example of a Cypher query can be seen in Listing 3.2. 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 3.3.

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

Listing 3.2: Simple matching example in Cypher

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

Listing 3.3: Equivalent matching query in SQL

A key feature of Cypher is the "ASCII art"-style syntax, 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 (22), see Section 2.2. Data modification is also possible in Cypher (68), using basic clauses such as `CREATE` to create new vertices 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.

3.2.3 G-CORE

G-CORE is a combined effort of academia and industry to steer the future of graph query languages (14). Angles et al. identify three main challenges with existing graph query languages. Firstly, current implementations output tables of values, vertices, or edges. Instead, the output should be graphs to improve usability so that the results of graph queries can again be queried. Secondly, G-CORE is unique in how 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. Therefore, they change the LPG model to a Path Property Model (PPG) in which paths are entities, in addition to vertices and edges, and can also have labels and properties. 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, the

goal of G-CORE is to capture the core of available languages and take what they do well to create a standard. The final goal is related to the fact that no standard existed when publishing the G-CORE paper in 2018.

3.2.4 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 (12). 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, including 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 vertex 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 (12).

3.2.5 GSQL

The graph database system TigerGraph was built to support massive parallel computation of queries and analytics (69). The high-level query language, GSQL, borrows syntax elements from plain SQL, procedural SQL dialects (such as PL/SQL and T-SQL), and G-CORE (14). It has strong imperative traits and is Turing complete (70), allowing users to express both graph queries and specify analytical graph algorithms. GSQL supports both a type-free approach where no schema is required, as well as a SQL-like strongly typed approach where the graph schema has to be defined prior to loading the graph in the database.

3.2.6 SPARQL

The query language SPARQL is designed for querying RDF, a directed labelled graph, data format, see Section 3.1. As of SPARQL 1.1, queries can involve filters, unions, value aggregations, nested queries, and path expressions (71). Queries in SPARQL are composed of triple patterns (*subject*, *predicate*, *object*) with different clauses and operators. The result

is a set of bindings, also referred to as a binding table (72). Systems such as Stardog (73) and Amazon Neptune (58) allow their data to be queried using SPARQL.

3.2.7 Gremlin

Rodriguez describes the Gremlin graph traversal language, developed and distributed by the Apache TinkerPop project (62). Gremlin's query approach consists of three main components, a graph, a traversal, and a set of traversers. The traversers move about the graph according to the instructions which the user can create. It is a functional programming language that can be implemented in other languages. However, this other language does need to support *function composition* and *function types.* Several key characteristics of Gremlin include that it supports both imperative and declarative pattern matching styles. Furthermore, it can be embedded in another programming language, and users can extend it. The traversal consists of a given number of steps arranged in either a linear or nested motif. With a linear motif, the output of each step forms the input for the next step. With the nested motif, the nested step(s) act as arguments for another step. The output of this step can then be used as input for the next step. The traversers continue to follow instructions for as long as possible. Whenever the traverser cannot follow the next traversal instruction, it halts. Once all traversers have halted, an answer to the traversal can be provided back to the user.

3.2.8 Graph Query Language

The Graph Query Language (GQL) is a standardised query language that is, at the time of writing in 2022, scheduled to be released in March 2024 (74). It will share the graph pattern matching language with SQL/PGQ (22). However, whereas SQL/PGQ is a read-only extension of SQL, GQL is a stand-alone standardised query language. GQL will have its Data Manipulation Language (DML) with Create, Read, Update, and Delete operations, as well as its Data Definition Language (DDL), such as Create Type and Create Graph (74). Another difference between SQL/PGQ and GQL is that GQL will support both schema-fixed and schema-flexible variants, whereas SQL/PGQ does not support schema-flexible graphs.

3.3 Mapping Strategies

3.3.1 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. (67). 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 (67). SQLGraph allows Gremlin queries to be converted into SQL (56). It uses a combination of relational and non-relational data storage. The *adjacency information* uses relational storage, while the attributes regarding all the vertices and edges are stored with JSON storage.

GraphGen (75) 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 (75).

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 (76). 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. (77), 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.

There is a question whether expressing graph algorithms using recursive SQL can be as fast as a direct implementation of these algorithms in a language such as C++.

3.3.2 Mapping from RDF to PG

Thakkar et al. (78) 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 (79) 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.

However, SPARQL engines tend to have lower performance than SQL databases due to the query optimisationand exectuion challenges posed by many self-joins (80) 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 (78). 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 (72). 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.

3.4 Graph Traversal Algorithms

Table 3.1 shows an overview of some algorithms that can be used for path-finding in various scenarios depending on the number of sources and destinations, as well as whether the graph is unweighted/weighted (yielding shortest/cheapest problems, respectively).

	Shortest path	Cheapest path
Single-source Single-destination	Bidirectional BFS	Bidirectional Dijkstra
Single-source All destination	BFS	Dijkstra Bellman-Ford
Multi-source All destination	Multi-source BFS	Multi-source Bellman-Ford Multi-source Dijkstra
All source All destination	All-source BFS	Floyd-Warshall

Table 3.1: Variatons of a graph traversal algorithms for various scenarios

3.4.1 Unweighted Shortest Path

With an unweighted shortest path, the goal is to find the path between a source vertex and a destination vertex with a 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 $\mathcal{O}(|V| + |E|)$, indicating that the time scales linearly to the number of vertices and edges in the graph (81).

Then et al. observed that performing graph analytical algorithms on large datasets using BFS as described above is time-consuming and involves redundant computation when executed multiple times from different vertices. Therefore they introduce Multi-Source BFS (37). A core challenge in creating efficient graph analytical algorithms was the lack of data access locality, causing many random accesses (82). MS-BFS is designed to run multiple BFSs over the same graph using a single CPU core. MS-BFS is designed to (1) share computation between BFSs; (2) reduce the number of random memory accesses; (3) not incur synchronisation costs. MS-BFS is most efficient in graphs with the small-world network property (83) since these graphs are well-connected. This property means that a BFS will discover most vertices in a few iterations, allowing other BFSs to have a high chance of discovering the same vertices in the same iteration, allowing access to a single

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
```

vertex to be shared among various BFSs. This reduces the number of random accesses and avoids redundant computation, reducing the overall run-time (37).

An advantage of the MS-BFS algorithm by Then et al. is that it can utilise a direction-optimised variant introduced by Beamer et al. (84). In the standard BFS algorithm, new vertices are discovered in a *top-down* manner, meaning that vertices are discovered by exploring the neighbours of the vertices found in the previous iteration. The direction-optimised variant introduces a *bottom-up* approach that scans for vertices that have yet to be discovered. A heuristic can be used every iteration to decide which approach to use. In practice, the top-down approach will be used in the earliest iterations, while the bottom-up will be used in the final iterations. This is because by then, most vertices have already been discovered and starting from the unseen vertices will result in fewer edges that need to be traversed.

Research has also been conducted on designing BFS algorithms for multi-core processors (85, 86). Chhugani et al. (87) present a scalable BFS algorithm using lock- and atomic-free operations. They observe that graph traversal algorithms do not fully utilise compute or bandwidth resources since the limit is memory latency (87). Each step of a BFS graph traversal involves iterating over a bounded set of vertices, looking at the adjacent vertices and updating the *Depth* and *Parent* arrays. It is the accesses to the adjacent

vertices and the *Depth* and *Parent* arrays that involve cache, and Translation Lookaside Buffer (TLB) misses. Therefore, the authors introduce an atomic-free update mechanism and rearrange the bounded set of vertices to reduce the TLB misses for accessing the adjacent vertices, making the algorithm no longer limited by memory latency. Additionally, the authors highlight optimisations to reduce the number of accesses to the *Depth* and *Parent* arrays using bit-arrays to denote whether a vertex has already been visited or not. They introduce a generalised, cache-resident, atomic-free representation of this bit-array (87).

Hong et al. present a parallel BFS algorithm for multi-core CPUs that uses randomly shaped real-world graph instances (86). The newly introduced BFS implementation performs better than state-of-the-art (85) as the size of the graph grows larger. Furthermore, the authors introduce a dynamic execution method for each BFS iteration that can choose between a sequential execution, a multi-core CPU execution, and a GPU execution.

Akiba et al. (88) propose a BFS algorithm that performs pruning during the search to reduce the search space referred to as *pruned landmark labelling.* The algorithm allows not to traverse edges from a vertex if the correct distance can be given using other, already labelled vertices. This method is most effective on highly central vertices in complex networks (89) since those vertices allow for the most pruning of other vertices.

3.4.2 Cheapest Path (Weighted Shortest Path)

For the cheapest path problem, commonly referred to as the weighted shortest path, the goal is to find the path with the minimum sum of edge weights from a source vertex to a destination vertex. The two textbook algorithms used are Dijkstra's algorithm and Bellman-Ford to compute the cheapest path in weighted graphs. In both of these algorithms, the graph is explored from a single source. However, these algorithms differ in how the graph is explored.

In Dijkstra's algorithm, seen in Algorithm 2, a graph and a single source vertex are 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 implemented using 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

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 vertices in Graph
 8:     while Q is not empty do
            u := vertex 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
```

source to this vertex is known. Dijkstra's algorithm has the constraint that there can only be non-negative edge weights. An optimisation for Dijkstra's algorithm is implementing the priority queue as a Fibonacci heap (90).

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 ∞ 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, a cheaper path is found, and the distance is updated.

If all vertices have been iterated 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, cheapest paths

Algorithm 3 Bellman-Ford

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

that are longer (i.e. consist of more edges) are found, and only at the last iteration is the cheapest path guaranteed to be found.

Dijkstra's algorithm using a Fibonacci heap has a worst-case time complexity of $\mathcal{O}(|E| + |V| \cdot \log |V|)$ (91), which is better than Bellman-Ford's worst-case time complexity of $\mathcal{O}(|V| \cdot |E|)$ (92). However, the expected runtime of Bellman-Ford is $\mathcal{O}(|E|)$ in large dense graphs with low diameter (93).

Then et al. (82) 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. The structures of these heaps are different, limiting the possibilities of using SIMD instructions. Based on experimental results, the authors report that the performance of the batched Bellman-Ford algorithm is 3–10× higher compared to batched Dijkstra's algorithm (82). Therefore, it was decided to implement the batched Bellman-Ford algorithm proposed by Then et al., as shown in Algorithm 6.

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 array will contain the distances from the source to all other vertices.

The batched Bellman-Ford implementation makes use of an optimisation introduced by Yen (93) that only checks the neighbours of a vertex if their distance was modified. Klein (94) introduces a multi-source shortest path algorithm for planar graphs with non-negative edge weights in $\mathcal{O}(|V| \cdot \log |V|)$. Cabello et al. present a generalisation of the

work by Klein in which the shortest path distance for multiple sources can be calculated in embedded graphs (95).

Algorithm 4 Floyd-Warshall

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

Both Dijkstra's algorithm and Bellman-Ford are single-source algorithms. Floyd-Warshall (96) (FW), 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.

FW uses a matrix with the dimensions $|V| \cdot |V|$ where $|V|$ is the number of vertices in the graph. During initialisation, the distances where the source is equal to the destination (i.e. the cells in the diagonal of the matrix) 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, $\mathcal{O}(|V|^3)$, equals the best-time complexity.

Penner and Prasanna provide a more cache-efficient implementation (97). However, while the FW algorithm is efficient on *dense* graphs to calculate all pairs shortest path, other algorithms, such as a variation of Dijkstra's algorithm presented by Demetrescu and

Italiano (98), are faster on *sparse* graphs. Han et al. (99) introduce a program generator for the FW algorithm using tiling, loop unrolling, and SIMD vectorisation.

3.5 Survey on Graph Database Management Systems

Since the field of graph processing workloads is quite scattered (5, 100), many different GDBMSs exist, having different strengths and weaknesses. Besta et al. (9) 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 and Section 3.1 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. The survey compares features and languages supported by the systems discussed. 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 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.

3.6 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.

3.6.1 Neo4j

Neo4j is a JVM-based NoSQL graph database (101, 102), mainly focused on transactional workloads (103). Query execution follows a conventional model and uses the Volcano optimiser generator (104). Query compilation makes use of the tuple-at-a-time iterator-based execution model. Alternatively, the query is compiled Java bytecode using a push-based execution model (11, 105).

Neo4j provides multiple ways to query graphs. First is the low-level core API used for elementary graph operations. The second uses the graph query language Cypher described in Section 3.2.2 (102). Complex queries can be better optimised using the core API and will often lead to better performance than Cypher (27).

3.6.2 Umbra

Umbra is a general-purpose disk-based (SSD-based) DBMS developed at the Technische Universität München (TUM) (106). Umbra claims to support arbitrary data sizes by using external memory whenever required. Furthermore, it retains performance when the working set fits inside the main memory by introducing a novel buffer manager (106). It is the successor of the in-memory system HyPer, a hybrid system able to handle both OLAP and OLTP simultaneously (107).

Similarly to HyPer, Umbra uses a compiled query execution strategy. The logical query plans are translated to parallel machine code (106) which calculates the query results. In case the query contains steps that can be multi-threaded, a morsel-driven approach (108) is used to distribute the work.

Umbra is compared to its predecessor HyPer and the column store MonetDB (109) using the TPC-H benchmark at scale factor 10. It is shown that Umbra has a 1.8× geometric mean speedup compared to HyPer and a 2.3× speedup compared to MonetDB. The pure in-memory performance of Umbra is comparable to that of HyPer (106). As Umbra is, in principle, able to use external memory whenever the working set no longer fits in main memory, it would be interesting to observe the performance on larger scale factors.

3.6.3 TigerGraph

TigerGraph is a graph database system tailored to support massive parallel computations of queries and analytics (69). Queries can be expressed using GSQL, which has been inspired by G-CORE (14), see Section 3.2.5.

TigerGraph is not an in-memory database system; the disk is used if the full graph does not fit in memory. The storage and processing engines have been implemented in C++ to allow for more fine-grained control over memory management. TigerGraph can automatically partition the graph across a cluster of servers to maintain performance. All edges from a given vertex are stored on the same server. Furthermore, TigerGraph has a distributed query mode in which all servers work on a query on an as-needed basis (69).

3.6.4 GRADOOP

Junghanns et al. present *GRADOOP* (110), 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 (110). On the other hand, parallel graph processing systems such as Google Pregel (111) 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 vertices 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 produce them, 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 (112). A brief demonstration of the system is provided. However, no actual results are presented, making it difficult to estimate the performance of GRADOOP.

3.6.5 The Case Against Specialised Graph Analytics Engines

Fan et al. (6) 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 (6). The users can interact with Grail through an API since formulating graph analytical queries in SQL may not feel natural to users. 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 (111) and GraphLab (113), which is mainly designed for parallel machine learning algorithms, are used to compare the performance of Grail. Three queries are tested, 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.

3.6.6 GRainDB

Jin et al. present GRainDB (18), 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. Consequently, GRainDB is not available in current DuckDB versions.

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 (WCOJ) algorithms. Furthermore, the authors extended SQL to support vertices, edges, and path patterns using inspiration from Cypher and GSQL.

3.6.7 MillenniumDB

Vrgoč et al. present MillenniumDB, a persistent, open-source, graph database (114). 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 (114). The engine is based on relational database management systems techniques, WCOJ (115) 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, such as WCOJ. Third, MillenniumDB should be easily extensible. 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 (116). They compared their performance with three RDF engines:

Jena TDB (117), Blazegraph (118), and Virtuoso (119), as well as the property graph database Neo4j Community Edition (101). It finds that the execution times of graph pattern matching queries are consistently faster for MillenniumDB than other database engines.

4

Design & Implementation

The goal of this thesis is, by building on the work of Singh et al. described in Section 2.4, to support shortest path-finding and returning the path length, as well as returning the *any* shortest path containing a list of vertices found on the shortest path. In addition, the SQL/PGQ implementation is also extended in DuckDB by adding support for path-finding on weighted graphs. Supporting weighted graphs requires us to extend the CSR creation as explained in Section 2.4.2 to support weighted edges and introduce a new scalar UDF to compute the cheapest path length.

This chapter will describe key components related to implementing efficient path-finding operators in DuckDB using C++. Figure 4.1 shows a schematic overview of the components with the new elements highlighted in blue.

4.1 Overview

This thesis extends the previous work in multiple ways. In Section 4.2, we explain how support is added to compute the shortest path length using the MS-BFS algorithm, see Section 4.2. In Section 4.3 an explanation is given on how the *any shortest path* function returning a list of vertices found in the shortest path is implemented. In Section 4.4, 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 4.5. The output for the path-finding operations is used in step ⑦.

In step ③ and step ⑦, we perform two join operations that create the same hash table. These steps will use an optimisation introduced in Section 4.6 which allows for the hash

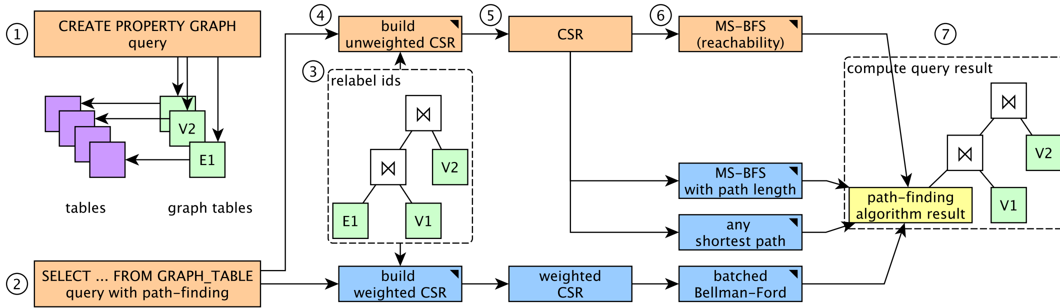


Figure 4.1: Schematic overview of key components for efficient path-finding operators

table used to perform join operations to be shared between multiple joins in case these joins are identical.

4.2 Shortest Path

Computing the shortest path between two vertices in an unweighted graph can be done using Multi-Source Breadth-First Search (MS-BFS) by Then et al. (37), pseudo-code is provided in Algorithm 5. MS-BFS is an extension of the textbook BFS discussed in Section 3.4. 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 can be common in SQL/PGQ. We seek to improve efficiency by sharing the memory access of multiple searches. Moreover, with MS-BFS, we can use the vectorised execution engine of DuckDB, possibly also parallelising the execution of MS-BFS. A bottleneck for the MS-BFS algorithm will likely be memory access (37). To alleviate the memory bottleneck, we create the CSR data structure on-the-fly before executing the MS-BFS algorithm.

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.

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.

Algorithm 5 Multi-Source Breadth-First Search

```
    Input: G, 𝔹, s
 2: seen_{s_i} ← {b_i} for all b_i ∈ 𝔹
    visit ← ⋃_{b_i∈𝔹} {(s_i, {b_i})}
 4: visitNext ← ∅
    while visit ≠ ∅ do
 6:     for each v in visit do
            𝔹'_v ← ∅
 8:         for each (v', 𝔹') ∈ visit where v' = v do
                𝔹'_v ← 𝔹'_v ∪ 𝔹'
10:         end for
            for each n ∈ neighbours_v do
12:             𝔻 ← 𝔹'_v \ seen_n
                if 𝔻 ≠ ∅ then
14:                 visitNext ← visitNext ∪ {(n, 𝔻)}
                    seen_n ← seen_n ∪ 𝔻
16:                 Do BFS computation on n
                end if
18:         end for
        end for
20:     visit ← visitNext
        visitNext ← ∅
22: end while
```

The shortest (unweighted) 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.

The implementation works with batch sizes to allow for the code to be SIMD-ised by the compiler. Due to the vectorised execution engine of DuckDB and the standard vector size being 1024 tuples, we will receive at most 1024 tuples (source vertices in this case) at a time. To process these more efficiently, we process these tuples in batches. The batch size can be changed, though we assume the batch size is set to 256 for the remainder of this thesis unless otherwise specified. Thus, a vector of 1024 vertices is processed in 4 batches. Within a batch, every unique source vertex gets assigned a *lane* for which we calculate the distance from this source to all other vertices in the graph.

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. There exist no SIMD instructions for 4-bit integers (49). A workaround is possible, though we do not consider that for this thesis. 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, it is possible to run $128/8 = 16$ BFS instances at once. With 32-bit integers we are only able to run $128/32 = 4$ BFS instances.

Figure 4.2 shows an example where 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 4.2, 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 4.3, 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 4.3b. We mark that the distance for these vertices 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 4.4 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.

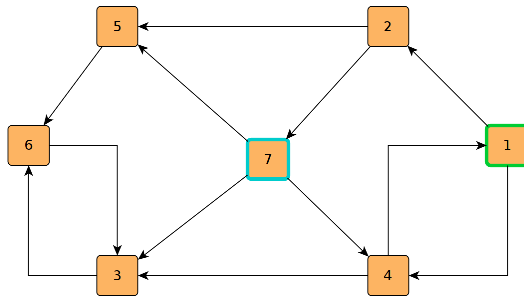
In step 3, shown in Figure 4.5, we discover the last vertices for both B1 and B2. At that point, we observe that no more edges lead to unseen vertices from these discovered vertices, and thus we terminate the algorithm.

4.3 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. We will only return a single one if there are multiple shortest paths of equal length. This algorithm will also use the MS-BFS described in Section 4.2. However, the MS-BFS algorithm needs to be extended to keep 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 $|V|$. 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 $|V|$. Thus, when reconstructing the path, if we reach $|V|$, 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

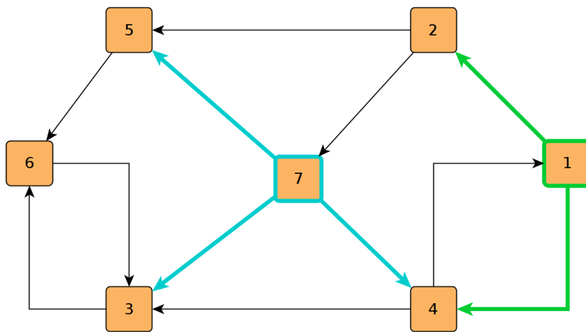


(a) Graph representation of the initial state

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 the initial state. The symbol **X** marks that the value of a bit is set to 1.

Figure 4.2: MS-BFS initial state

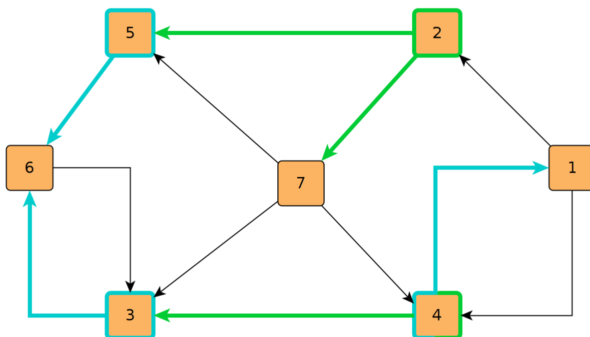


(a) Graph representation of step 1

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. The symbol **X** marks that the value of a bit is set to 1.

Figure 4.3: MS-BFS step 1

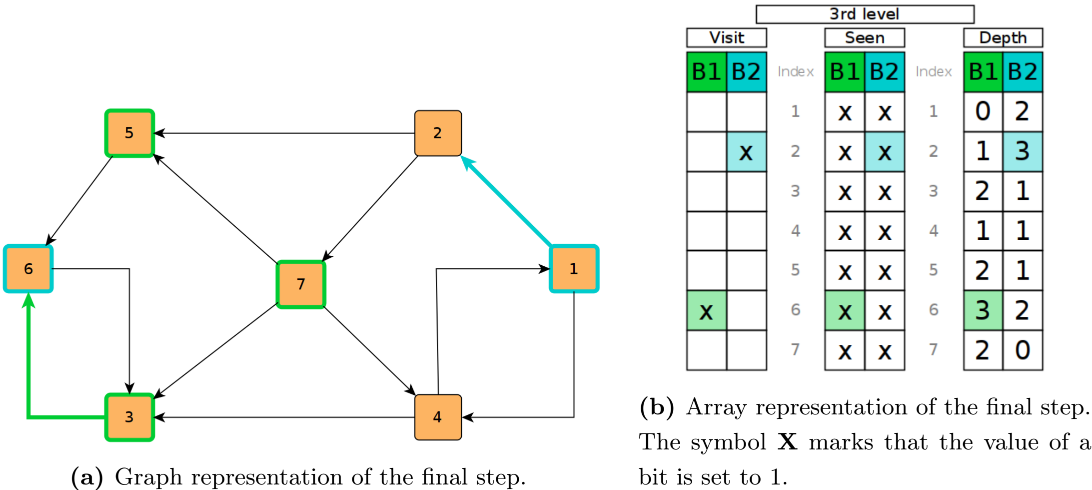


(a) Graph representation of step 2

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. The symbol **X** marks that the value of a bit is set to 1.

Figure 4.4: MS-BFS step 2



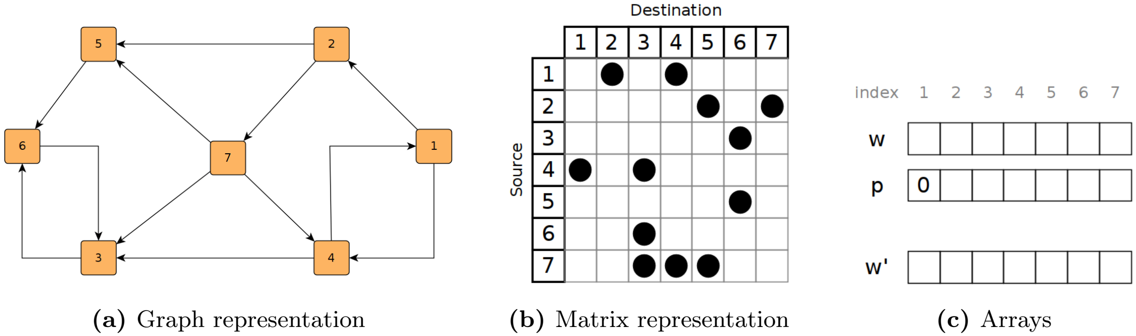
(a) Graph representation of the final step.

(b) Array representation of the final step. The symbol **X** marks that the value of a bit is set to 1.

Figure 4.5: MS-BFS step 3 (final)

always know that this is from a longer path, which is not interesting. Therefore, we set a boolean *is_set* to true to indicate that this value should not be changed later.

In 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.



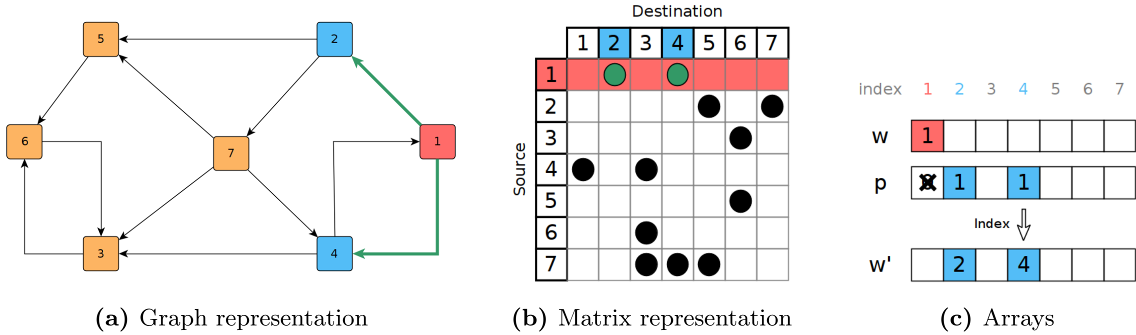
(a) Graph representation

(b) Matrix representation

(c) Arrays

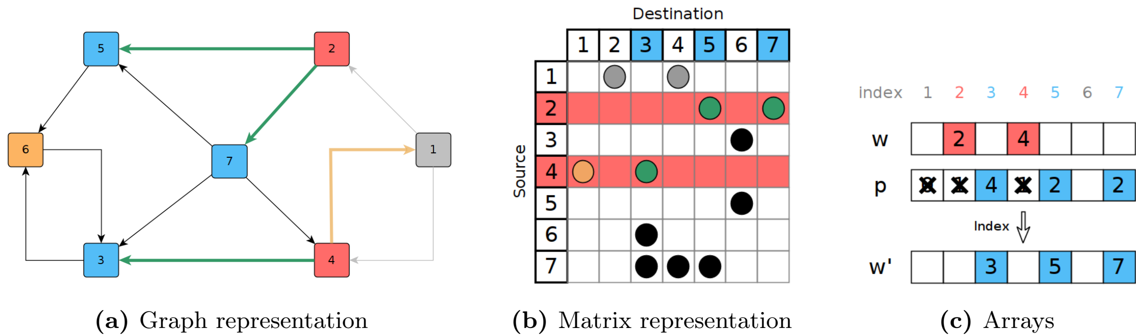
Figure 4.6: Any shortest path initial state

Constructing the paths Figure 4.6 shows the initial state where no algorithm steps have been taken. The source from which the algorithm starts is node 1. In this algorithm,



(a) Graph representation (b) Matrix representation (c) Arrays

Figure 4.7: Any shortest path step 1



(a) Graph representation (b) Matrix representation (c) Arrays

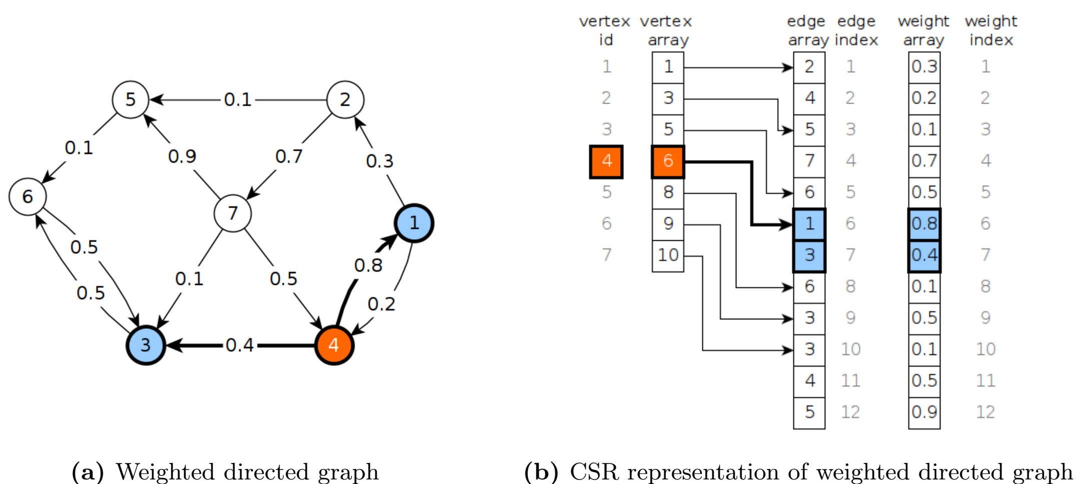
Figure 4.8: Any shortest path step 2

the destination node is only 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.

Figure 4.7 shows the first step, where node 1 is set as the active node in array w, seen in Figure 4.7c. 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 4.8, 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.

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



(a) Weighted directed graph

(b) CSR representation of weighted directed graph

Figure 4.9: Weighted directed graph and the CSR representation

Returning the paths In DuckDB, it is possible to return a column of type List (120). 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. This is done by inserting 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 shown in the example is single-source. However, the implementation in DuckDB uses the MS-BFS algorithm to traverse the graph, making the *any* shortest path function multi-source.

4.4 Weighted Compressed Sparse Row

In order to compute the cheapest path, we need to extend the CSR data structure to support edge weights. Supporting edge weights does not require significant modification to the data structure. The only addition is an array that stores the weight related to an outgoing edge, see Figure 4.9, similar to the edge array described in Section 2.4.2. The current implementation in DuckDB supports 64-bit integers and doubles to be stored as

edge weights. Storing the weights in the CSR is the same way the edges are stored. If we want to create a CSR with the weights, the UDF responsible for creating the edge array receives an additional variable, namely the column containing the weights, allowing the CSR to be still created in parallel. Other data types that take up fewer bits should also be supported for future work.

4.5 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 3, 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 to evaluate multiple source-destination pairs simultaneously. 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. (82) 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. The structures of these heaps are different, limiting the possibilities of using SIMD instructions. Based on experimental results, the authors report that the performance of the batched Bellman-Ford algorithm is 3–10× higher compared to batched Dijkstra's algorithm (82). Therefore, we decided to implement the batched Bellman-Ford algorithm proposed by Then et al., as shown in Algorithm 6. The implementation also handles the sources within a vector in batches as described in Section 4.2

4.5.1 Batched Bellman-Ford implementation

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 ∞ to indicate

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
```

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 updated $dists[n][i]$ accordingly.

A possible optimisation was introduced by Yen (93) 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 ∞, there is no chance it will provide a cheaper distance as $\infty + weight > \infty$.

We expect Bellman-Ford to be slower compared to the MS-BFS algorithm by Then (37). 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, in the worst case, we have to perform $|V| - 1$ iterations until we have found all the cheapest paths. With Bellman-Ford, we cannot give this guarantee after one iteration since it can be the case that a cheaper path is discovered later. Only once all vertices have been iterated over and no change has been made can we guarantee that all cheapest paths have been found. This fact requires us to perform another iteration over all vertices. 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.

4.5.2 Vectorising Batched Bellman-Ford

Batched Bellman-Ford, as shown in Algorithm 6 will not be auto-vectorised by the clang or GCC compilers[1]. This is due to the modified array and the if-statement in line 19 (if *newDist* != *dists*[*n*][*i*] then . . .), which creates a branch. This alone does not necessarily prevent vectorisation (121). 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 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.

[1] `https://godbolt.org/z/zz7WeMshE`
[2] `https://gcc.godbolt.org/z/MfnMcb8he`

We include this implementation of the algorithm without the use of a modified array for comparison sake.

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.

4.6 Shared Hash Join

Queries can contain multiple joins that are identical. Identical joins can occur in graph-like queries, where the vertex table is joined twice on an edge table: once for the edge's source and once for the its destination, see Listing 4.1, where `knows` is the edge table and `person` is the vertex table.

```
1 SELECT *
2 FROM knows k
3 JOIN person src ON k.person1_id = src.person_id
4 JOIN person dst ON k.person2_id = dst.person_id;
```

Listing 4.1: Example of identical joins

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. This occurs during the creation of the CSR data structure described in Section 2.4.2, see lines 21 and 22 of Listing A.1 in order to label the edges as `(src, dst)` combinations of vertex table row ids. In this case, the right side is identical in both joins. See Figure 4.10 for the relevant part of the physical plan of this example query.

An optimisation is to build the 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

[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
```

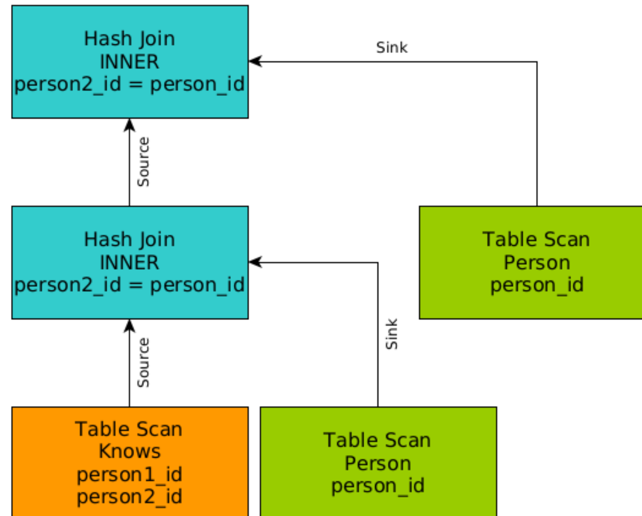


Figure 4.10: Example of a duplicate sink state in the physical plan

table multiple times. We decided to add this optimisation to DuckDB for the benefit of CSR construction performance but also the benefit of any other queries containing any repeated identical joins. 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.

We have implemented this optimisation as follows. Our initial idea for the implementation was to detect duplicate joins in the optimiser. The optimisation can be done by adding a new rule to the optimiser, which scans for duplicate joins. In the optimiser, the logical plan tree is 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 will 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 turned out to be several challenges with this approach. First, using only the name of the left-hand side table to detect duplicate joins is not 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. However, this can also be done in the optimiser.

Assuming the duplicate join is detected, a logical node would have replaced this join. However, this requires the introduction of a new logical node and a new physical operator for the physical planner stage. This new physical operator would then have created its own pipeline, creating additional work that could be saved using our chosen approach.

The chosen approach is to detect the duplicate joins during the pipeline creation, later in the execution pipeline than the previous approach. At this point, the physical plan has already been created and gets converted into pipelines. When the physical operator `HASH JOIN` is detected, we save the corresponding pipeline that builds the hash table, and the `HASH JOIN` operator is saved in a dictionary inside the *Executor*. Then, whenever an additional `HASH JOIN` is detected, we check if this join results in a duplicate hash table as 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. If the `HASH JOIN` is found to be equal, we replace its pipeline that will build a hash table with a reference to the original pipeline. Hence, we prevent the duplicate join from building the same hash table and instead potentially reusing the original hash table multiple times.

A disadvantage of the current implementation is that it only works for syntactically completely equal joins. Subsumed joins or joins with different column orders are not supported.

5

Evaluation

5.1 Experimental Setup

5.1.1 Environments

We conducted several experiments to assess the scalability of the proposed algorithms and the effectiveness of the shared hash join optimization. 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: 32 KiB per core (data), 32 KiB per core (instr), L2 cache: 256 KiB per core, L3 cache: 2.5 MiB per core; shared
- 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: 32 KiB per core (data), 32 KiB per core (instr), L2 cache: 256 KiB per core, L3 cache: 2 MiB per core; shared[1]
- Compiler: GCC 9.4.0
- Compiler options for SSE2: `-O3`
- Compiler options for AVX2: `-O3 -mavx2`
- Operating system: Ubuntu 20.04.4 LTS

[1] `https://en.wikichip.org/wiki/intel/core_i7/i7-8650u`

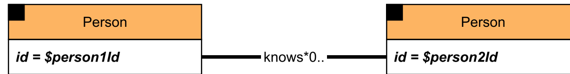


Figure 5.1: LDBC Interactive Query 13 (1)

5.1.2 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 (122). The benchmark contains datasets structured similarly to real-world social networks, consisting of persons and their connections. These datasets exhibit natural social-network phenomena such as the small-world property (83). 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 larger the scale factor, the more vertices and edges are contained in the graph.

For every query, the benchmark has generated parameters on which the performance can be evaluated. The parameters are generated so that they all provide similar run-time behaviour (122).

Interactive query 13. Interactive query 13 (1), see Figure 5.1 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. We should return -1 as the distance if no path is found. 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 (83), any two people will likely have a path containing six or fewer hops.

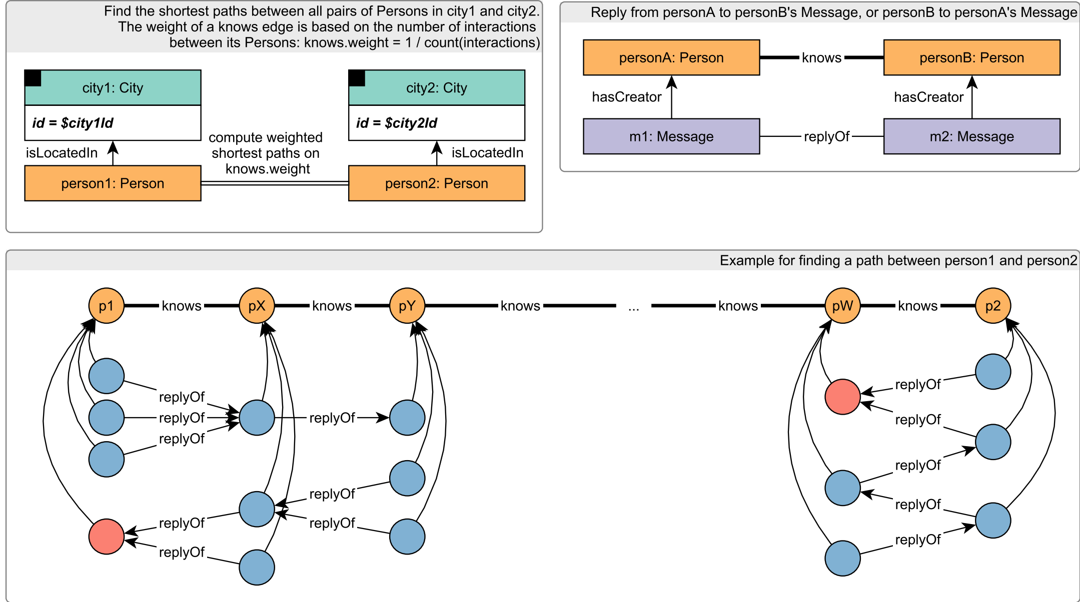


Figure 5.2: LDBC BI Query 19 (2)

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 $(\text{person1id} + \text{person2id})\%10 + 1$.

Two queries require us to find the shortest or cheapest path from some starting point to a target. These queries are 19 (2), and 20 (3).

BI query 19. For BI query 19 (2), see Figure 5.2, 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. For this query, there are two sets of parameters generated by LDBC. For the first set of parameters, referred to as 19a from now, `City1` and `City2` are small cities within the same `Country`. For the second set of parameters, 19b, `City1` and `City2` are small cities from different `Countries`. People from the same country have a higher chance of knowing each other than persons from different countries. Therefore, q19a will have more source-destination pairs compared to q19b. We use 19a to evaluate the performance.

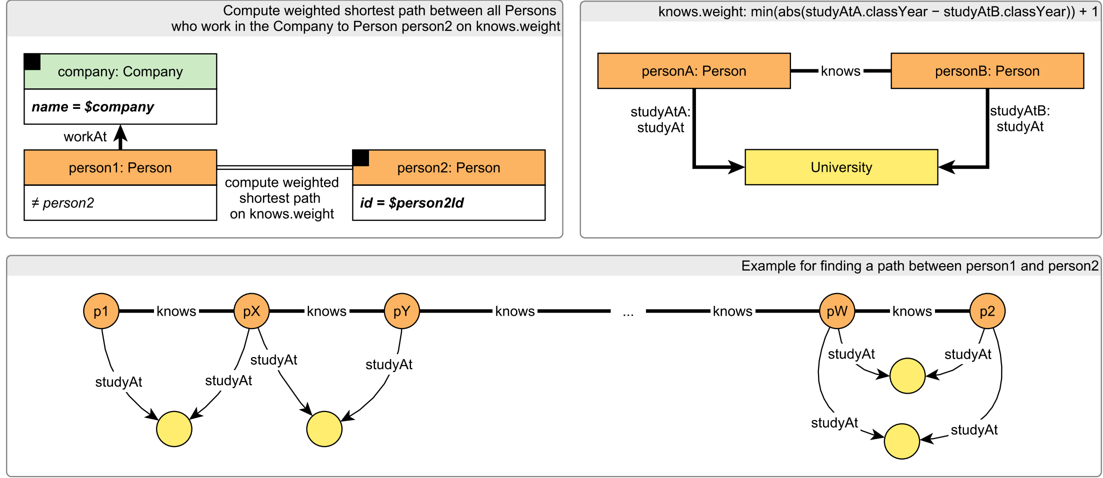


Figure 5.3: LDBC BI Query 20 (3)

BI query 20. BI query 20 (3), see Figure 5.3, 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 4.5. If all weights were equal, MS-BFS, discussed in Section 4.2 could be used.

The queries are executed in several phases. In the first phase, DuckDB performs any pre-computing required for the query. For example, for BI query 20, the edge weights are pre-computed and stored in an *edge* table. This edge table only contains entries that are relevant to the query. For example, for BI query 20, any edge of `personA` who knows a `personB` that has not studied at the same university as `personA` will be filtered out since this edge is not relevant to computing the path length. The total search space is reduced by filtering, 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. This phase is followed by inserting parameters into a temporary table. It is then possible to compute the shortest or cheapest paths for all rows in this temporary table and return the result.

Scale factor	Number of vertices	Number of edges	Number of src-dst pairs	Shortest path length run-time (s) / src-dst pair	Cheapest path length run-time (s) / src-dst pair
1	9 538	346 028	377	0.0023	0.1881
3	23 486	1 057 792	378	0.0036	0.5908
10	65 141	3 678 708	385	0.0079	1.8123
30	162 139	11 048 604	388	0.0185	6.3378
100	453 895	37 311 030	395	0.0561	26.1644
300	1 147 729	111 313 830	400	0.1487	70.5652
1 000	3 298 534	374 495 576	389	0.6568	285.0050
3 000	8 692 445	1 118 720 370	390	1.8107	N/A
10 000	25 633 648	3 709 057 850	359	8.3948	N/A

Table 5.1: Graph size and number of source-destination pairs per scale factor for Interactive query 13

5.2 Shortest Path Length

Table 5.1 shows the number of source-destination pairs evaluated in query 13 for every scale factor. In Interactive query 13, the number of source-destination pairs remains relatively constant as the scale factor increases.

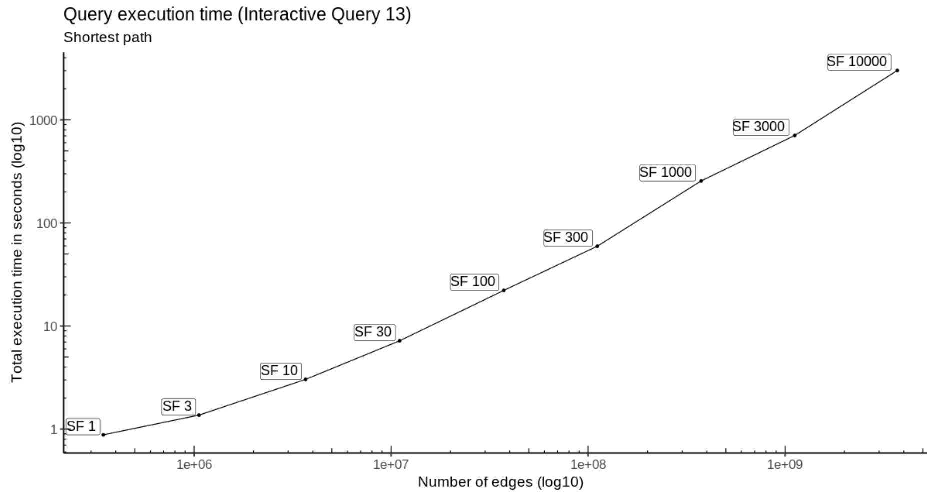


Figure 5.4: Average execution time per scale factor for shortest path Interactive query 13

In Figure 5.4 we observe the average run-time for Interactive query 13 using the shortest path length 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 in the graph. 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. In the largest scale factor, 10 000, we search for 400 path lengths in a graph with three billion *knows* edges, this was done in 3 013 seconds ($\approx$ 50 minutes).

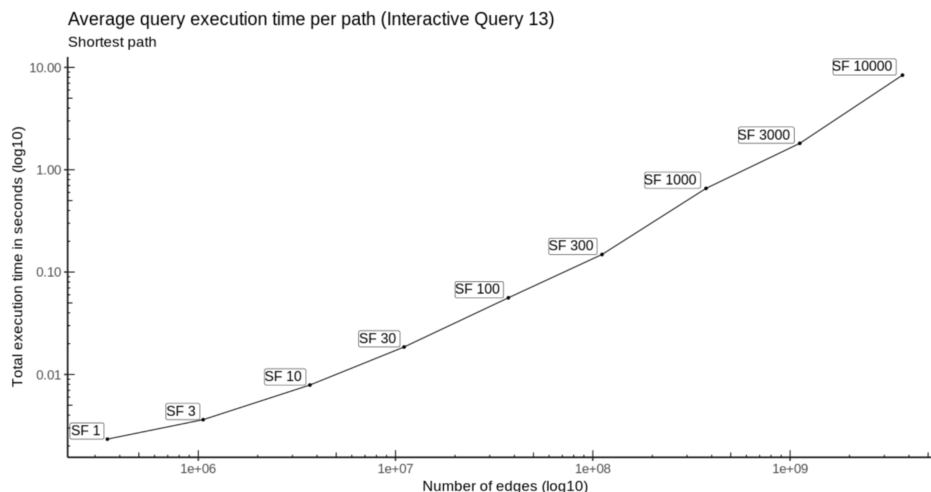


Figure 5.5: Average execution time per source-destination pair per scale factor for shortest path Interactive query 13

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 the number of seconds. We observe that the algorithm uses more time for every source-destination pair on average as the scale factor increases. For scale factor 10 000, it takes 10 seconds to find the shortest path length for a source. The increasing time taken makes sense; as the search space increases, more edges must be traversed until all shortest paths are found.

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 larger 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 a constant amount of time, roughly 0.3 seconds.

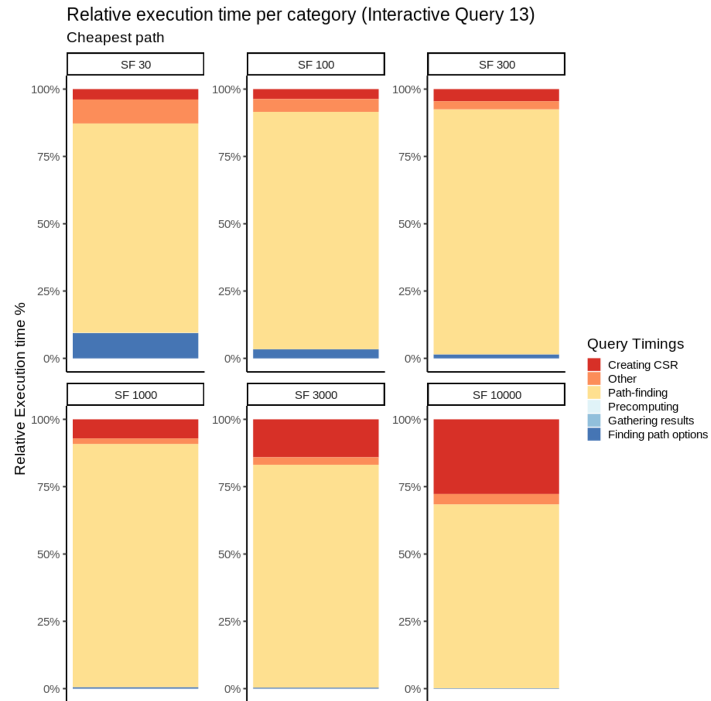


Figure 5.6: Relative time spent per phase for Interactive query 13 shortest path

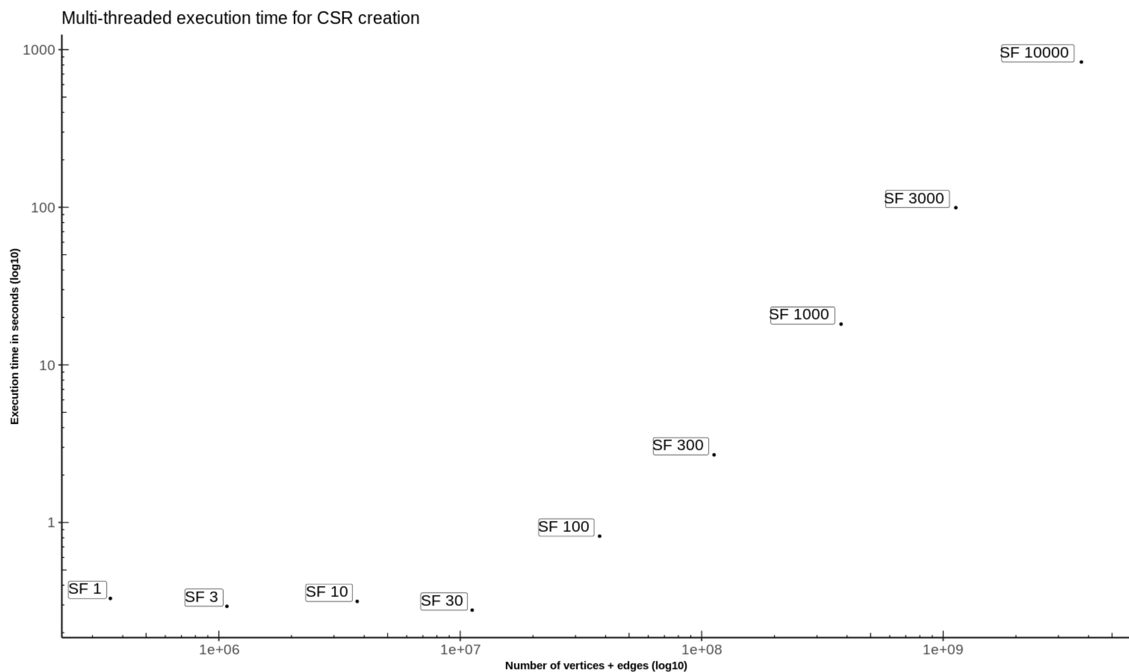


Figure 5.7: CSR creation time for all scale factors

However, we see a linear increase in the time it takes to create this data structure for larger-scale factors. This is because, with the smaller scale factors, it is possible to perform the creation in parallel. However, at some point, there are no more threads to which a morsel can be given. Recall that a work is scheduled on a thread for every $120 \cdot 1024 = 122\,880$ tuples. The environment in which this experiment was conducted has 96 threads. Hence, once we exceed $122\,880 \cdot 96 = 11\,796\,480$ tuples, all threads are filled. At scale factor 30, there are 162 139 vertices and 11 048 604 edges, which are fewer tuples than can be handled with 96 threads. However, at scale factor, there are 453 895 vertices and 37 311 030 edges, which result in too many tuples to handle in parallel. Therefore, from scale factor 100, we observe a linearly scaling for the time taken to create the CSR data structure.

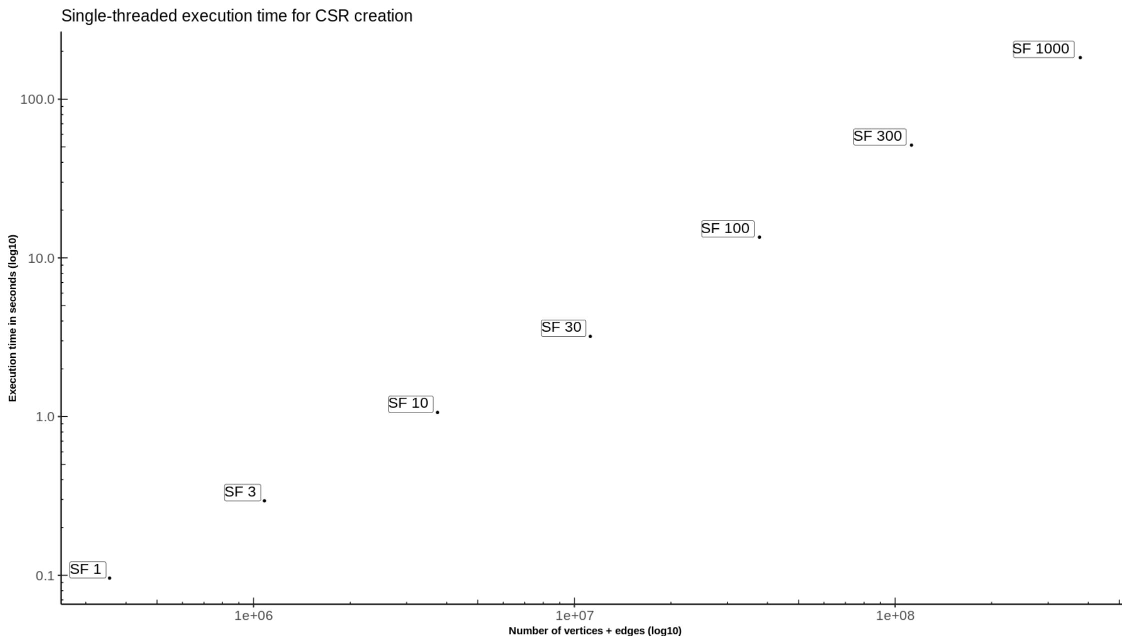


Figure 5.8: CSR creation using a single thread

Figure 5.8 shows the execution time for creating the CSR data structure using a single thread. This confirms that the time taken to create the CSR data structure scales linearly with the number of edges and vertices in the graph. Intuitively this makes sense as the CSR creation, as described in Section 2.4.2, is performed with one loop over all vertices $|V|$ and one loop over all edges $|E|$.

5.3 Cheapest Path Length

5.3.1 Interactive Query 13

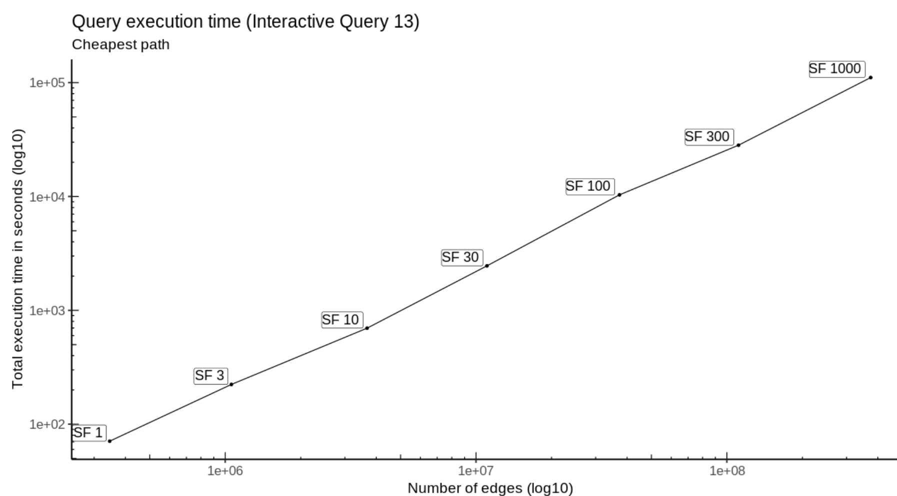


Figure 5.9: Total execution time per scale factor for the cheapest path variant of Interactive query 13

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

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.

5.3.2 Comparison Between Shortest and Cheapest Path Length

We conducted a performance comparison between the shortest and cheapest path length functions using Interactive query 13.

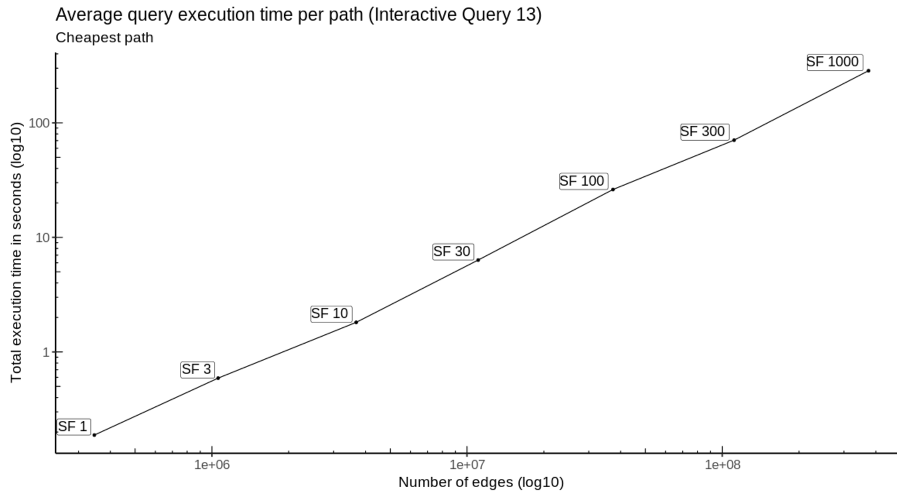


Figure 5.10: Average execution time per source-destination pair per scale factor for the cheapest path variant of Interactive query 13

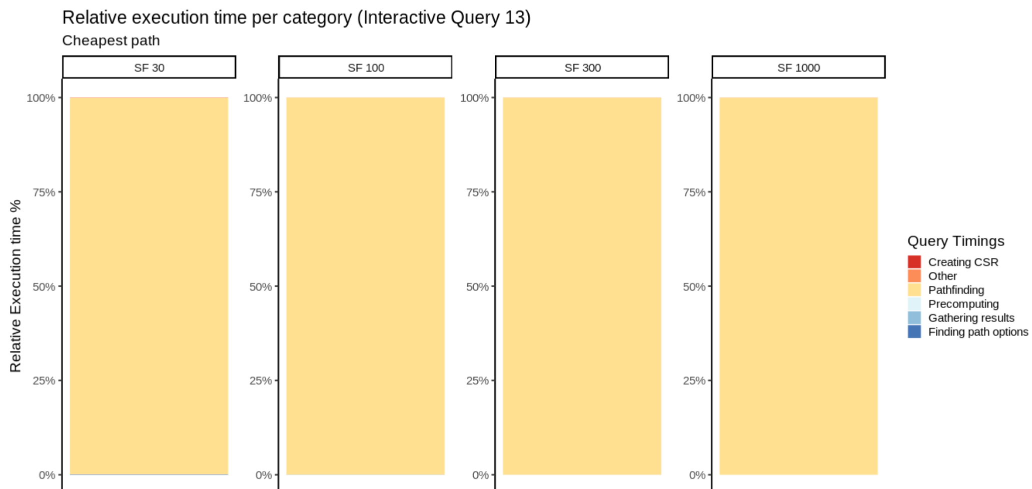
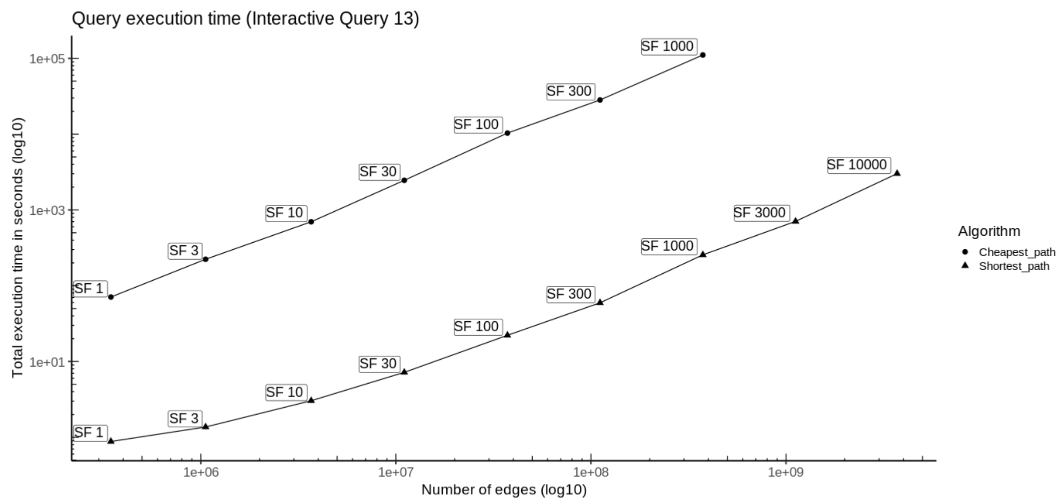


Figure 5.11: Relative time spent per phase for Interactive query 13 cheapest path



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

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

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

Figure 5.12: 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 5.12a. Since the y-axis uses a logarithmic scale, Figure 5.12b has been included to make comparison easier. The shortest path function is at most 475× faster.

The large performance gap can likely be attributed to three factors. The first is that the problem of computing the cheapest path length is more difficult than the shortest path length. Azad et al. (123) have created a comparison of the execution time for various algorithms between graph frameworks. They observe that, depending on the dataset, BFS is 2-10x faster compared to Single-source shortest path using the delta-stepping algorithm.

Second is that Bellman-Ford is not the most-efficient algorithm, even though it fits well into the DuckDB execution model and can make good use of SIMD instructions (82). However, Bellman-Ford is work-inefficient for graphs with only non-negative edge weights, whereas Dijkstra's algorithm or delta-stepping would be more efficient (124, Section 4.2).

The third factor is that the implementation introduced performance bottlenecks, mainly due to the use of dictionaries. Dictionaries make use of a hash table for the key. Thus, every time we wish to access a value in the hash table, a hash value needs to be calculated first. This leads to a pointer which is a random memory access.

Due to all these factors, it is expected that batched Bellman-Ford is slower than MS-BFS. However, by eliminating the third factor, we should see the performance gap be decreased substantially. Further optimisations are covered in Chapter 6.

If the graph is larger than can be stored in memory, this causes cache misses every iteration. Following is a more in-depth analysis of the space required to calculate the shortest and cheapest path lengths. The cache sizes in the environment tested are:

- L1 cache: 32 KiB per core (data), 32 KiB per core (instr)
- L2 cache: 256 KiB per core
- L3 cache: 2.5 MiB per core; shared

CSR creation The CSR structure in DuckDB uses 64-bit integers to store the vertices, edges, and weights in the case of a weighted graph. In the scale factor 1 data set, there are 9 538 vertices and 346 028 edges. Thus the CSR structure requires $9\,538 \cdot 64 = 76$ KiB for the vertices, $346\,028 \cdot 64 = 2\,768$ KiB $= 2.7$ MiB for the edges, and $346\,028 \cdot 64 = 2\,768$ KiB $= 2.7$ MiB for the weights. Thus, not all CSR data can be stored in the L2 cache on a single core for even the smallest scale factors. L3 cache can be used as that is shared between the cores and is larger than L2.

Shortest path length For the shortest path length function, there are three bitmaps of length $|V|$. In total there are 400 unique sources, however these are handled in batches of 256 elements (lanes) at a time. Thus, the first batch handles the first 256 unique sources. The second batch handles the remaining 144 sources. Thus the maximum total space the three bitmaps used was $3 \cdot 9\,538 \cdot 256 = 915$ 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 256 unique sources required $256 \cdot 9\,538 \cdot 8 = 2\,441$ KiB. Therefore, the total size for the shortest path function is $2\,441 + 915 + 2\,768 + 76 = 6\,200$ KiB $= 6.05$ MiB. This data does not fit inside L2 cache, though it does fit in the shared L3 cache.

Cheapest path length For computing the cheapest path length, the distances are stored in a 64-bit integer matrix. For every source, a new lane is created of length $|V|$. When testing with 400 unique sources in batches of 256 elements at a time, this matrix has a total size of $256 \cdot 64 \cdot 9538 = 19\,533$ KiB $= 19.5$ MiB. Additionally, there is a bitmap

[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

Scale factor	Number of vertices	Number of edges	Number of source-destination pairs	Run-time (s) / source-destination pair
1	9 322	346 028	2 185	0.1846
3	23 002	1 057 792	12 878	0.4824
10	63 988	3 678 708	89 650	1.7033

Table 5.2: Graph size and number of source-destination pairs per scale factor for BI query 19a

of length $|V|$ to keep track of the modified entries, which takes $256 \cdot 9\,538 = 305$ KiB. The maximum total size required for computing one batch of the cheapest path length is: $305 + 19\,533 + 2\,768 + 2\,768 + 76 = 25\,450$ KiB = 25.4 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 even in scale factor 1.

For larger scale factor, more L3 cache needs to be used and at some point even L3 will fill up. The situation is slightly better for the shortest path length function compared to the cheapest path length function since it requires less total storage. 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.

5.3.3 BI Query 19a

Table 5.2 shows the size of the graph and the number of source-destination pairs evaluated per scale factor. Different from what was shown in Table 5.1, in the case of BI query 19a, the amount of source-destination pairs increases with the larger scale factors. Computing the cheapest path length for more source-destination pairs becomes increasingly difficult.

Figure 5.13 shows the average run-time for BI query 19a using the cheapest path length function. Note that the x-axis and y-axis are a logarithmic scales. The largest scale factor evaluated was scale factor 10. The scaling appears to be linear, though the absolute performance and scalability are lacklustre. Since the number of edges for BI query 19a and Interactive query 13 being similar, it is likely that the absolute performance drop is due to there being more source-destination pairs for BI query 19a, compared to Interactive query 13, increasing the total time needed to calculate all cheapest path lengths.

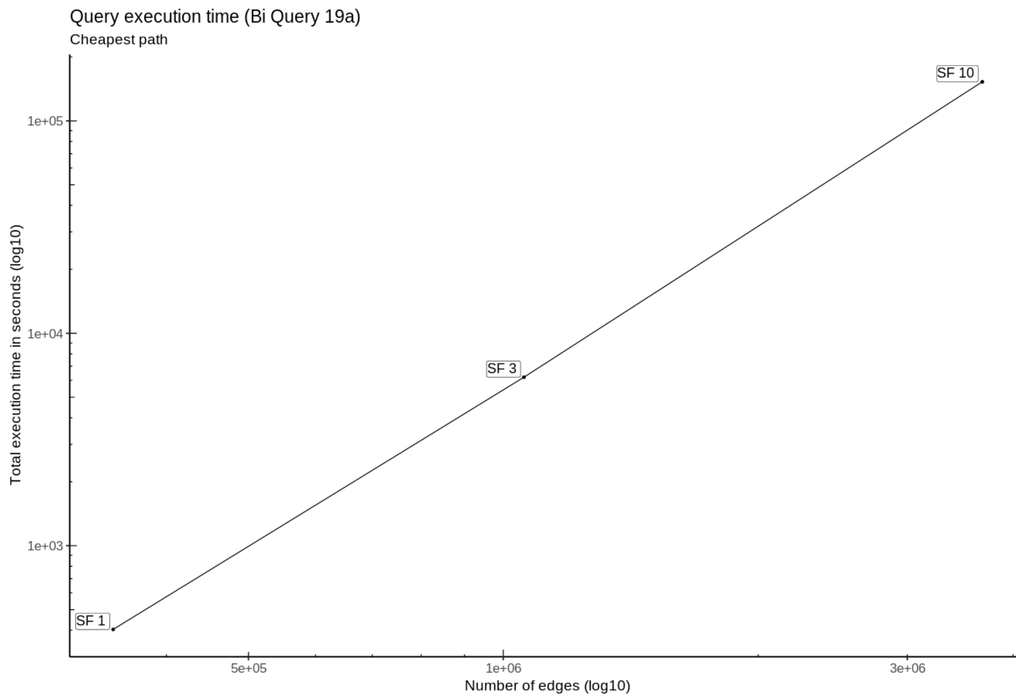


Figure 5.13: Average execution time per scale factor for cheapest path BI query 19a

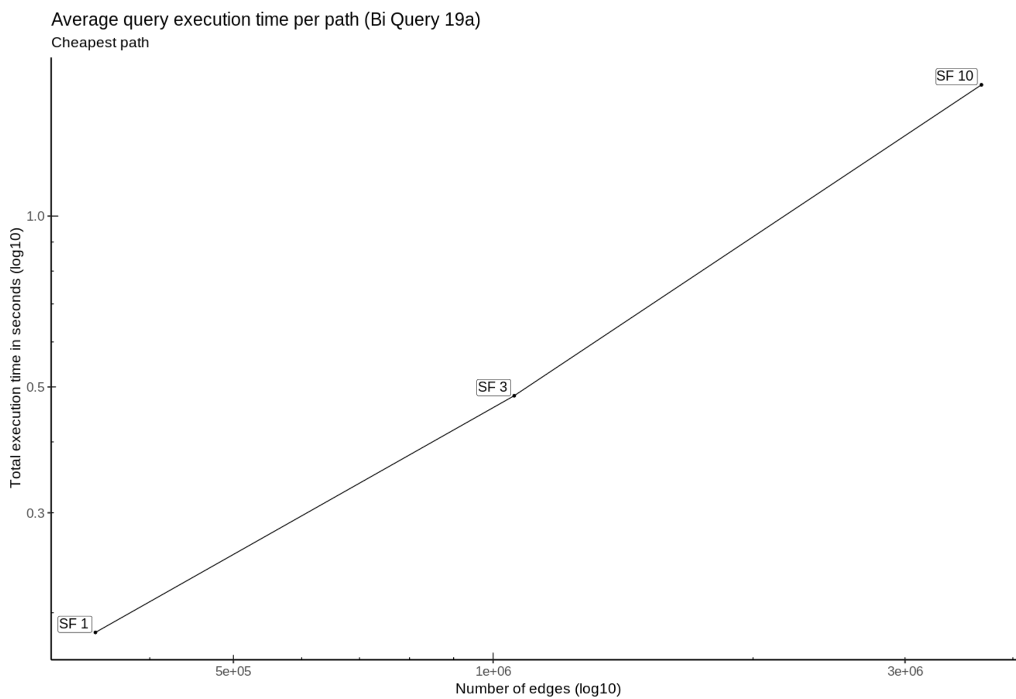


Figure 5.14: Average execution time per source-destination pair per scale factor for cheapest path BI query 19a

Figure 5.14 shows the average time spent per source-destination pair. The x-axis and the y-axis are logarithmic scales. We observe that the time spent to calculate the cheapest path length for a single source-destination pair increases with the number of edges in the graph. For scale factor 10, it takes over one second to calculate the cheapest path length.

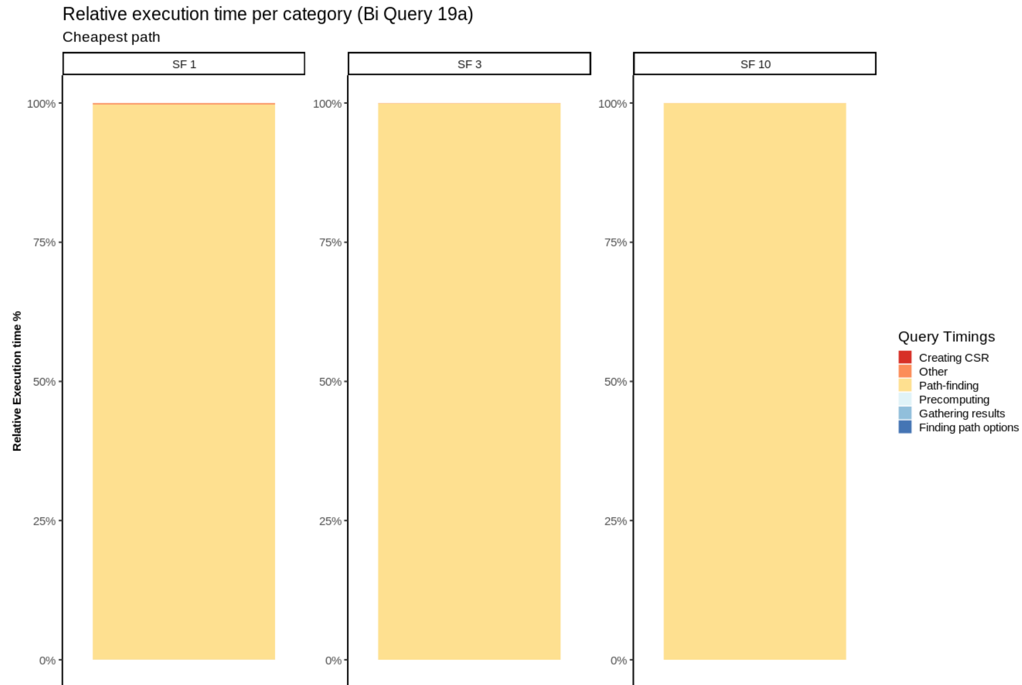


Figure 5.15: Relative time spent per phase for cheapest path BI query 19a

Figure 5.15 shows the relative time spent in each phase of the query. Similar to Figure 5.6, we observe that the path-finding dominates all other categories.

5.3.4 BI Query 20

Table 5.3 shows the size of the graph, the number of source-destination pairs and the average run-time per source-destination pair per scale factor for BI query 20. Compared to BI query 19 shown in Table 5.2, there are roughly 3 times fewer vertices, and between 25 and 11 times fewer edges depending on the scale factor. Furthermore there are less source-destination pairs to evaluate, leading to a lower average run-time per source-destination pair.

Figure 5.16 shows the average run-time for computing the cheapest path length for BI query 20. As the graph is smaller compared to BI query 19a, the scalability is marginally

Scale factor	Number of vertices	Number of edges	Number of source-destination pairs	Run-time / source-destination pair (in seconds)
1	3 017	13 732	198	0.0046
3	8 623	62 324	1 269	0.0050
10	27 693	321 794	7 791	0.0090
30	78 677	1 333 952	19 543	0.1508
100	255 062	6 171 262	66 328	0.5579

Table 5.3: Graph size and number of source-destination pairs per scale factor for BI query 20

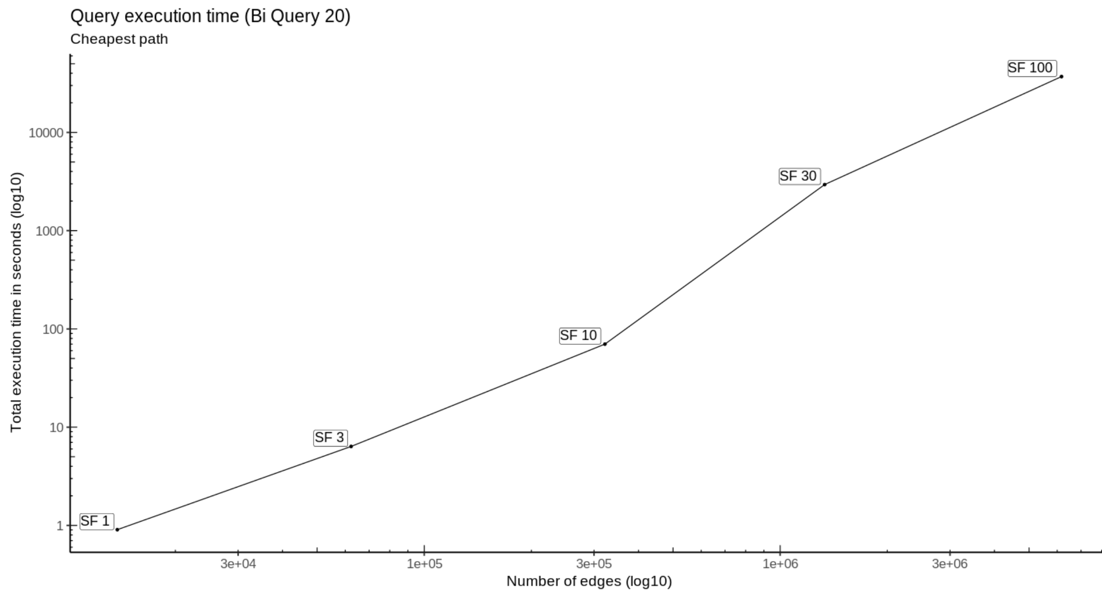


Figure 5.16: Average execution time per scale factor for cheapest path BI query 20

better for BI query 20. This can be attributed to the smaller graph sizes as well as a lower amount of source-destination pairs that need to be evaluated.

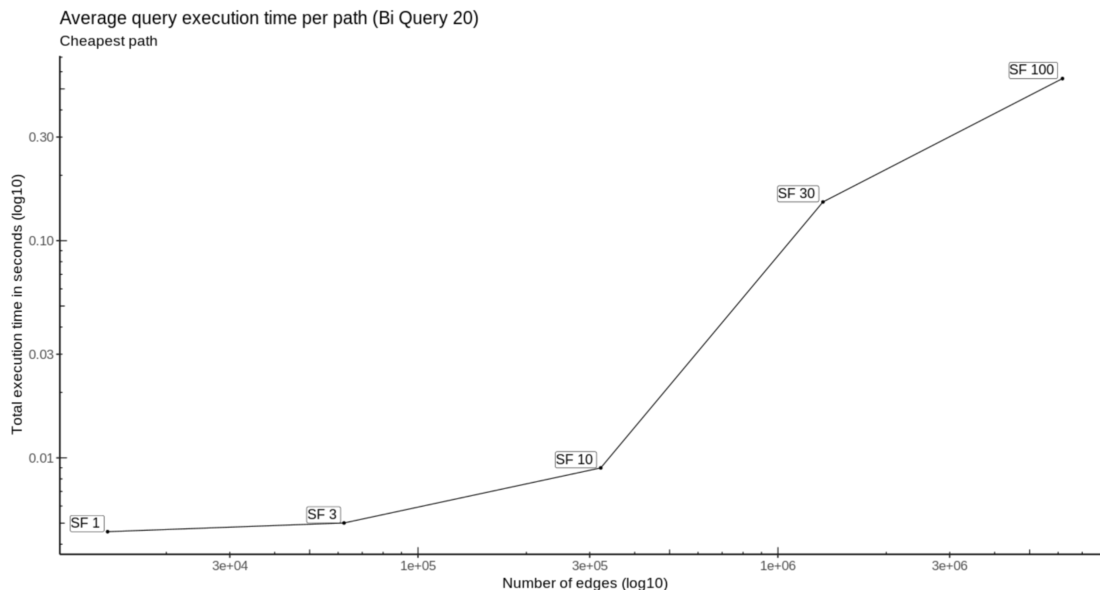


Figure 5.17: Average execution time per source-destination pair per scale factor for cheapest path BI query 20

Figure 5.17 shows the average run-time required per source-destination pair. Whereas Figure 5.14 for BI query 19a showed a linear increasing run-time for the smaller scale, BI query 20 shows a small increase in average query execution time between scale factor 1 and scale factor 10. The increase of average execution time between scale factors 10 and 30 is steeper.

Figure 5.18 shows the relative time spent in each phase of the query. It can be observed that in BI query 20, relatively more time is spent precomputing the vertices and edges, compared to Interactive query 13 and BI query 19a. However, for the larger scale factors, computing the cheapest path length dominates the other phases of the query similar to what was observed in Interactive query 13 and BI query 19a.

5.4 Vectorised Batched Bellman-Ford

To assess the performance of an auto-vectorised implementation of the batched Bellman-Ford algorithm, we have conducted a microbenchmark, shown in Figure 5.19a. We conduct a microbenchmark before integrating the auto-vectorised batched Bellman-Ford in DuckDB since it is not always guaranteed that vectorising an algorithm improves the performance.

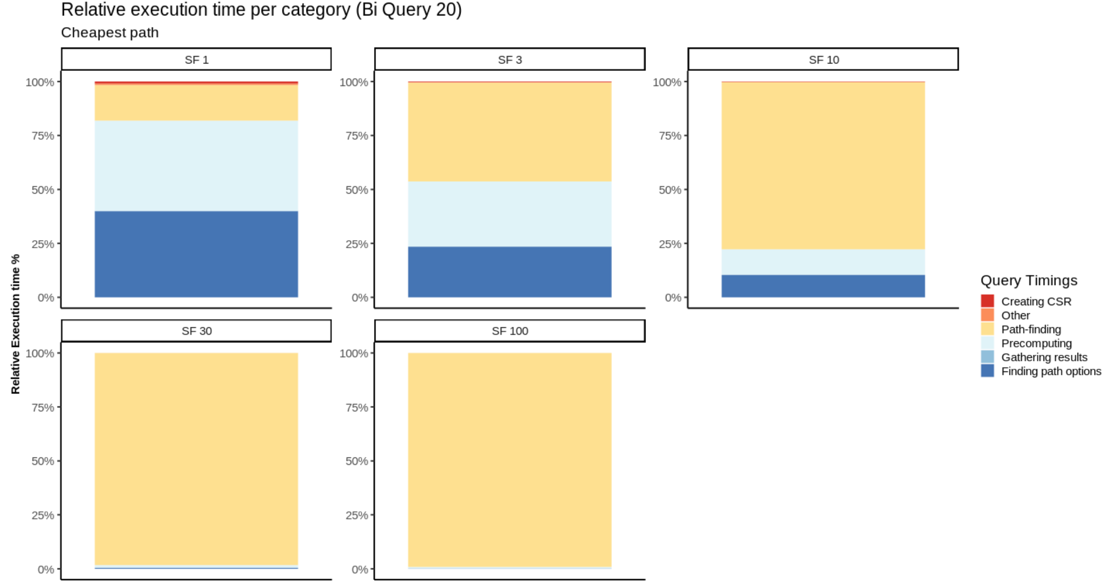


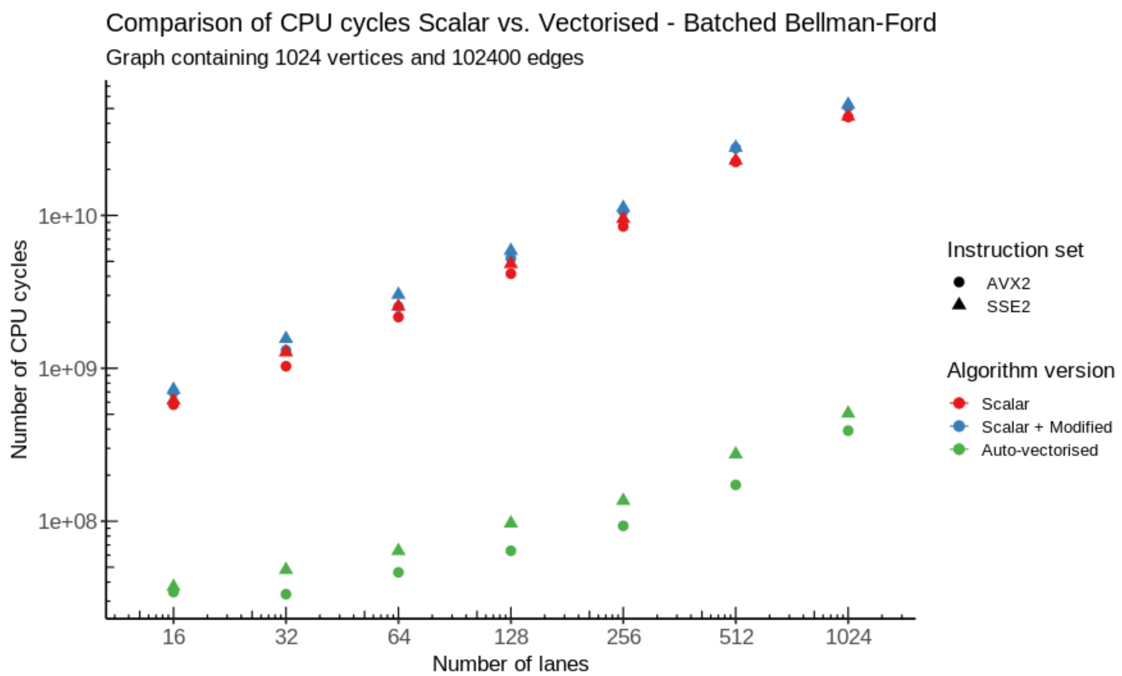
Figure 5.18: Relative time spent per phase for cheapest path BI query 20

The microbenchmark seen in Figure 5.19a is performed on a tiny 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 in the cache. This allows for a more accurate assessment of the influence of the SIMD instructions compared to the scalar instructions.

In Figure 5.19a 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 optimisations. The second one is also a scalar implementation with the added optimisation of the modified array used to avoid checking unnecessary neighbours. The third is the auto-vectorised implementation. We have also performed experiments on both the SSE2 and AVX2 instruction sets.

As shown in Figure 5.19a, the auto-vectorised 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 5.19b to highlight the speedup across the various lanes between the scalar and the vectorised AVX2 implementation. We observe that when we have 16 lanes, the vectorised AVX2 version is 16 times faster. The most considerable speedup is achieved when there are 512 lanes, in which case the vectorised version is 129 times faster. The speedup is lower when doubling the number of lanes to



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

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

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

Figure 5.19: Performance of scalar vs. auto-vectorised implementations of batched Bellman-Ford

1024, in which case the speedup is 112x.

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 in the environment tested are:

- L1 cache: 32 KiB per core (data), 32 KiB per core (instr)
- L2 cache: 256 KiB per core
- L3 cache: 2 MiB per core; shared

Thus, in the smallest case the total size is $4 + 400 + 64 = 468$ KiB. Therefore, this is too much for L1 and L2; however, it does fit in L3. 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 speed-up shown in Figure 5.19b 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 simultaneously. When there are more lanes, the average amount of active lanes for an active vertex increases. If only a single lane is active, it is expensive to visit the neighbours, as this is random memory access. However, the memory cost of visiting the neighbours gets amortised with the higher number of lanes.

5.5 Shared Hash Join

To evaluate the effectiveness of the shared hash join optimisation, DuckDB will execute the query in Listing 5.1 twice; once with the optimisation enabled, once with the optimisation disabled. We will be using the `person` and `knows` tables from the LDBC SNB dataset. The `person` table is always smaller in size compared to the `knows` table.

```
1 SELECT * FROM knows k
2     JOIN person src ON k.person1_id = src.person_id
3     JOIN person dst ON k.person2_id = dst.person_id;
```

Listing 5.1: Query used to evaluate the shared hash join optimisation

In theory, optimisation should result in faster joins, as time is saved by not building the hash table used for the join a second time. Recall that a join consists of both building a hash table and probing this hash table. The hash table will be built based on the smaller `person` table. The larger `knows` table will then be used to probe the hash table.

The larger the `knows` table is compared to `person` table, the more time is spent probing the hash table relative to building the hash table. Hence, we test the optimization with various table sizes for `knows` relative to the `person` table. The impact of not building a second hash table should be more noticeable when the tables are of similar size.

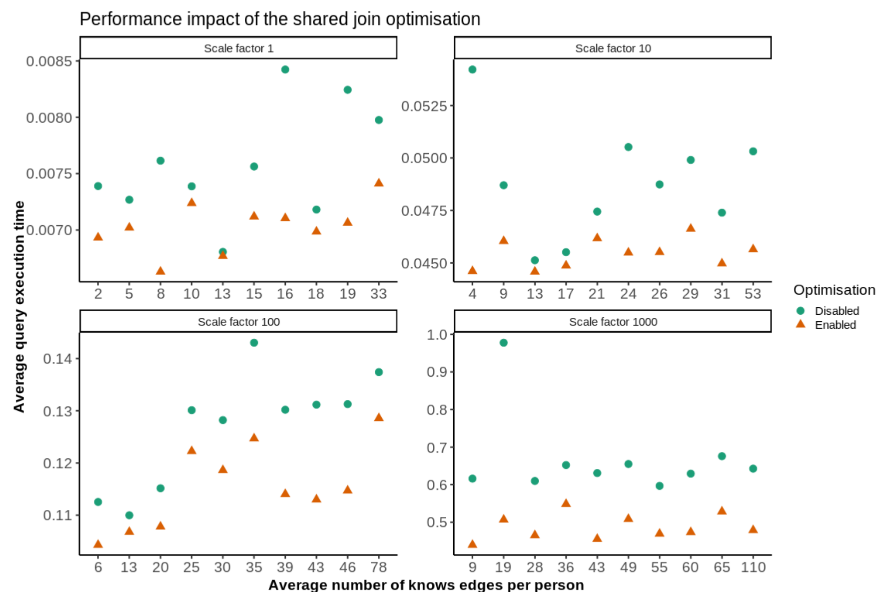


Figure 5.20: Performance comparison of the shared hash join optimisation

Figure 5.20 shows the average query execution time with the shared hash join optimisation enabled and disabled. The x-axis shows the average number of `knows` edges per person. The more `knows` edges, the larger the size of the `knows` table is compared to the `person` table. Using the query from Listing 5.1 we observe that the query execution time is always lower with the optimisation enabled. However, the optimisation does not appear to save more time when the average number of `knows` edges is lower, i.e. when the

`person` and `knows` tables are more similar in size. A possible explanation is that the time probing the hash table dominates the time to build the hash table. To determine this, conducting further microbenchmarks is necessary. Another observation is that the query execution times stay constant, whereas the size of the `knows` table increases.

6

Future Work

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

We have observed that the MS-BFS implementation is able to scale well. A further optimisation for the MS-BFS algorithm is to perform a bidirectional search. In this optimisation, exploration would be started from 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 expensive, it is crucial 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. A bitmap of length $|V|$ can be used to keep track of the sources, setting the bit to 1 whenever a unique source is encountered. If the bit is already 1, we know a search has been done in a previous batch or will be done in the current batch. When the bit is 0, this is a unique source; hence we flip the bit and need to perform a search from this vertex. Currently, every new batch in the current implementation will overwrite the results from the previous batch. To further limit the number of searches, the batch results should be saved, as these can contain source vertices for a later batch. If the same source is found in a later batch, the result from an earlier batch can be used instead.

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.

The current implementation determines the number of lanes (batch size) at compile time. An optimisation is to template the number of lanes and have variations for 1, 4, 16, 64, 256, and 1024 lanes, such that the ideal amount of lanes can be selected during run-time.

For example, if there are just 8 sources in a vertex, it is wasteful to use 1024 lanes, as a large majority will be empty.

A limitation for MS-BFS and batched Bellman-Ford is the random memory access to fetch neighbours. These random accesses are unavoidable; however, to mitigate the latency, it is possible to perform *neighbour prefetching*. We *prefetch* information of neighbours before we need them. This optimisation is performed by Then et al. in their MS-BFS implementation and shows an improvement by up to 25% (37).

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 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.

An optimisation related to the vectorised batched Bellman-Ford algorithm, see Section 4.5 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.

At the start, we will use 8-bit unsigned integers to store the distances, allowing us to store distances up to 255. If it is known that the maximum weight is w, then there can be $255/w$ safe iterations before an overflow is possible. If an overflow can occur during the next iteration, 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.

For optimal use of SIMD instructions, the edge weight and the currently stored distances must be of the same data type, so no data conversions need to be performed. Furthermore, an optimisation should also use the smallest data type required 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.

In DuckDB, the number of threads used for a query is decided during the table scan. A new thread is used for approximately every 120 000 tuples. However, as shown in the

experiments, the path-finding queries will not often contain more than 120 000 source-destination pairs 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. We can utilise the lanes better by ensuring that the vectors are full.

7

Conclusion

In this thesis, we aimed to create efficient path-finding operators that can be used in implementations of the upcoming SQL/PGQ standard scheduled to be released in June 2023 as a part of SQL:2023 (22). The path-finding operators have been implemented in the open-source RDBMS DuckDB. Following are answers to the research questions defined for this thesis work.

How to best implement path-finding algorithms in DuckDB? It was decided to implement the path-finding algorithms through a lightweight approach using scalar user-defined functions. Using scalar UDFs allows the integration of SQL/PGQ to be non-intrusive and be primarily limited to a DuckDB extension module. Nevertheless, still using the parallelisation and vectorised query execution model provided by DuckDB. Before every path-finding operation, a CSR data structure is created on-the-fly. Experiments show that creating the CSR data structure can be done in parallel and its creation time becomes small compared to the path-finding algorithms on large graphs.

How can path-finding best be implemented for unweighted graphs? In DuckDB, path-finding for unweighted graphs is implemented using the Multi-Source Breadth-First Search algorithm by Then et al. (37) designed to run multiple BFSs over the same graph using a single CPU core. Experiments on the LDBC Social Network Benchmark have shown that the largest scale factor available (SF10 000) can be handled, showing that the algorithm can scale well.

How can path-finding best be implemented for weighted graphs? Path-finding for weighted graphs in DuckDB is implemented using the batched Bellman-Ford algorithm

by Then et al. (82). The performance gap between the shortest and cheapest path-finding implementations is more significant than expected. The cheapest path-finding implementation contains performance bottlenecks that result in poor absolute performance compared to the shortest path-finding implementation.

What are the bottlenecks in the current SQL/PGQ implementation? A bottleneck identified in the current SQL/PGQ implementation was the creation of duplicate hash tables in the case where a query contains multiple identical joins. The duplicate joins result in time spent building the same hash table multiple times.

Another performance bottleneck is related to the current implementation of the batched Bellman-Ford algorithm in DuckDB. Experiments have shown that the performance of the cheapest path length function using batched Bellman-Ford is 80–475× slower than the shortest path length function using MS-BFS. Azad et al. (123) have observed that the BFS algorithm calculates the shortest path 2–10× faster tha the delta-stepping algorithm used to calculate the cheapest path on graphs with identical structure. In addition, experiments have shown that the query execution time is heavily dominated by calculating the cheapest path length. Therefore, the performance of the batched Bellman-Ford implementation is seen as a performance bottleneck.

How can these bottlenecks be optimised? Since the hash tables are identical, we have implemented an optimisation which detects these duplicate joins and ensures that a hash table is only built once. If a duplicate join builds the same hash table, it will reuse the hash table instead, saving the time it takes to construct it.

The performance of the path-finding operations can be optimised by (1) minimising the total number of searches by only searching from unique searches, (2) performing neighbour prefetching, (3) using the smallest the data type required to allow for more lanes to fit in a SIMD register. More optimisations are described in Chapter 6.

What is the performance impact of vectorisation on the path-finding algorithms? Our microbenchmark has shown that an auto-vectorised implementation of the batched Bellman-Ford algorithm can be, depending on the number of lanes, up to 129× faster than an implementation that does not make use of SIMD instructions. The speedup can be attributed to (1) making use of SIMD instructions and, (2) amortising the memory cost of visiting the neighbour vertices when there are more lanes.

To conclude, in this thesis we have shown that implementing path-finding operations to support SQL/PGQ in an existing RDBMS is feasible. We implemented our operations as a lightweight extension to the DuckDB RDBMS. We assessed the scalability of the implementation, confirming that the proposed approach can take advantage of the vectorised execution model of DuckDB. Furthermore, we identified a number of performance optimisations including both algorithmic improvements (e.g. bidirectional search) and implementation techniques (e.g. improved usage of SIMD instructions).

References

[1] Linked Data Benchmark Council. **LDBC SNB Interactive complex read query 13**. https://ldbcouncil.org/ldbc_snb_docs/interactive-complex-read-13.pdf, 2022. Accessed: 13/06/2022. vi, 64

[2] Linked Data Benchmark Council. **LDBC SNB BI read query 19**. https://ldbcouncil.org/ldbc_snb_docs/bi-read-19.pdf, 2022. Accessed: 13/06/2022. vi, 65

[3] Linked Data Benchmark Council. **LDBC SNB BI read query 20**. https://ldbcouncil.org/ldbc_snb_docs/bi-read-20.pdf, 2022. Accessed: 13/06/2022. vi, 65, 66

[4] 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

[5] 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, 39

[6] 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, 42

[7] M. Tamer Özsu. **Graph Processing: A Panoramic View and Some Open Problems**. In *VLDB*, 2019. 1, 25

[8] 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, 8, 10

[9] 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, 7, 8, 39

[10] TigerGraph. **TigerGraphDB**. `https://www.tigergraph.com/`, 2022. Accessed: 02/02/2022. 1, 8

[11] 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, 27, 28, 40

[12] 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, 27, 29

[13] Linked Data Benchmark Council. **LDBC Council**. `https://ldbcouncil.org/`, 2022. Accessed: 04/02/2022. 2

[14] 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. 2, 3, 27, 28, 29, 41

[15] 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

[16] 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. 2

[17] 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, 26, 27

[18] Guodong Jin, Nafisa Anzum, and Semih Salihoglu. **GrainDB: A Relational-core Graph-Relational DBMS**. In *CIDR*, 2022. 2, 8, 26, 42

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

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

[21] 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

[22] 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, 10, 11, 28, 30, 89

[23] 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, 11, 12, 13

[24] 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

[25] 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. 3, 13

[26] 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. 4, 14

[27] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases, 2nd Edition*. O'Reilly Media, Inc., 2015. 8, 40

[28] Oracle Labs. **Oracle PGX**. `https://docs.oracle.com/cd/E56133_01/latest/index.html`, 2022. Accessed: 08/03/2022. 8

[29] Peter Boncz. **A Survey Of Current Property Graph Query Languages**. `https://homepages.cwi.nl/~boncz/job/gql-survey.pdf`, 2022. Accessed: 17/02/2022. 8

[30] Neo4j Query Languages Standards and Research Team. **GQL Scope and Features**. 12 2018. 8

[31] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009. 12

[32] 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. 12

[33] Marcin Zukowski. **Balancing vectorized query execution with bandwidth-optimized storage**. 2009. 13

[34] Goetz Graefe. **Volcano - An Extensible and Parallel Query Evaluation System**. *IEEE Trans. Knowl. Data Eng.*, **6**(1):120–135, 1994. 13

[35] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. **Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask**. *Proc. VLDB Endow.*, **11**(13):2209–2222, 2018. 13

[36] 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. 13

[37] 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. 15, 16, 33, 34, 46, 56, 86, 89

[38] Thomas Vilhena. **Index-free adjacency**. `https://thomasvilhena.com/2019/08/index-free-adjacency`, 2019. Accessed: 21/02/2022. 16

[39] Andrew Lumsdaine, Douglas P. Gregor, Bruce Hendrickson, and Jonathan W. Berry. **Challenges in Parallel Graph Processing**. *Parallel Process. Lett.*, **17**(1):5–20, 2007. 16

[40] Dean De Leo and Peter A. Boncz. **Teseo and the Analysis of Structural Dynamic Graphs**. *Proc. VLDB Endow.*, **14**(6):1053–1066, 2021. 17

[41] Laurens Kuiper. **Fastest table sort in the West - Redesigning DuckDB's sort**. `https://duckdb.org/2021/08/27/external-sorting.html`, 2021. Accessed: 15/08/2022. 19

[42] Intel. **Intel Launches the Pentium® III Processor**. `https://www.intel.com/pressroom/archive/releases/1999/dp022699.htm`, 1999. Accessed: 13/07/2022. 21

[43] 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. 21

[44] Steam. **Steam Hardware and Software Survey**. https://store.steampowered.com/hwsurvey/, 2022. Accessed: 05/07/2022. 21

[45] David Kanter. **Intel's Sandy Bridge Microarchitecture**. https://www.realworldtech.com/sandy-bridge/6/, 2010. Accessed: 13/07/2022. 21

[46] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 1*. Intel Corporation, August 2007. 22

[47] ARM. **Introducing NEON Development Article - ARM SIMD instructions**. https://developer.arm.com/documentation/dht0002/a/Introducing-NEON/What-is-SIMD-/ARM-SIMD-instructions, 2022. Accessed: 15/08/2022. 22

[48] ARM. **Neon**. https://developer.arm.com/Architectures/Neon, 2022. Accessed: 15/08/2022. 22

[49] Intel. **Intel® Intrinsics Guide**. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html, 2022. Accessed: 13/07/2022. 22, 48

[50] Google. **cpu_features library**. https://github.com/google/cpu_features, 2022. Accessed: 13/07/2022. 23

[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. 23

[52] Forrest Smith. **Memory Bandwidth Napkin Math**. https://www.forrestthewoods.com/blog/memory-bandwidth-napkin-math/, 2020. Accessed: 15/08/2022. 24

[53] F. Dahlgren, M. Dubois, and P. Stenstrom. **Sequential hardware prefetching in shared-memory multiprocessors**. *IEEE Transactions on Parallel and Distributed Systems*, **6**(7):733–746, 1995. 24

[54] 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. 25, 27

[55] 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. 25

[56] 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. 25, 31

[57] BlazeGraph. **BlazeGraphDB**. `https://blazegraph.com/`, 2022. Accessed: 02/02/2022. 25

[58] 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. 25, 30

[59] 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. 25

[60] Thomas Neumann and Gerhard Weikum. **The RDF-3X engine for scalable management of RDF data**. *VLDB J.*, **19**(1):91–113, 2010. 25

[61] Gabor Szarnyas. **Self-joins**. `https://szarnyasg.github.io/posts/self-joins/`, 2022. Accessed: 11/03/2022. 26

[62] Marko A. Rodriguez. **The Gremlin graph traversal machine and language (invited talk)**. In James Cheney and Thomas Neumann, editors, *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015*, pages 1–10. ACM, 2015. 27, 30

[63] Neo4j. **Neo4j**. `https://neo4j.com/`, 2022. Accessed: 29/06/2022. 27

[64] 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. 27

[65] 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. 27

[66] 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. 27

[67] 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. 27, 31

[68] 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. 28

[69] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. **TigerGraph: A Native MPP Graph Database**. *CoRR*, **abs/1901.08248**, 2019. 29, 41

[70] TigerGraph. **GSQL Graph Algorithm Library**. `https://docs-legacy.tigergraph.com/v/2.3/graph-algorithm-library`, 2022. Accessed: 28/06/2022. 29

[71] The W3C SPARQL Working Group. **SPARQL 1.1 Overview**. `https://www.w3.org/TR/sparql11-overview/`, 2022. Accessed: 07/06/2022. 29

[72] 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. 30, 32

[73] Kendall Clark, Mike Grove, and Evren Sirin. **Stardog**. `https://www.stardog.com/`, 2022. Accessed: 22/06/2022. 30

[74] Keith Hare. **Property Graph Standards, Process & Timing**. `https://ldbcouncil.org/event/fifteenth-tuc-meeting/attachments/keith-hare-property-graph-standards-process-and-timing.pdf`, 2022. Accessed: 08/08/2022. 30

[75] 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. 31

[76] Kangfei Zhao and Jeffrey Xu Yu. **Graph Processing in RDBMSs**. *IEEE Data Eng. Bull.*, **40**(3):6–17, 2017. 31

[77] 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. 31

[78] 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. 31, 32

[79] Eric Prud'hommeaux and Andy Seaborne. **SPARQL Query Language for RDF**. `https://www.w3.org/TR/rdf-sparql-query/`, 2022. Accessed: 07/07/2022. 31

[80] Spyros Kotoulas, Jacopo Urbani, Peter A. Boncz, and Peter Mika. **Robust Runtime Optimization and Skew-Resistant Execution of Analytical SPARQL Queries on Pig**. In Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*, **7649** of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2012. 32

[81] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001. 33

[82] 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. 33, 37, 54, 73, 90

[83] D. J. Watts and S. H. Strogatz. **Collective dynamics of 'small-world' networks**. *Nature*, (393):440–442, 1998. 33, 64

[84] Scott Beamer, Krste Asanovic, and David A. Patterson. **Direction-optimizing breadth-first search**. In Jeffrey K. Hollingsworth, editor, *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, page 12. IEEE/ACM, 2012. 34

[85] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. **Scalable Graph Exploration on Multicore Processors**. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*, pages 1–11. IEEE, 2010. 34, 35

[86] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. **Efficient Parallel Graph Exploration on Multi-Core CPU and GPU**. In Lawrence Rauchwerger and Vivek Sarkar, editors, *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*, pages 78–88. IEEE Computer Society, 2011. 34, 35

[87] Jatin Chhugani, Nadathur Satish, Changkyu Kim, Jason Sewall, and Pradeep Dubey. **Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency**. In *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*, pages 378–389. IEEE Computer Society, 2012. 34, 35

[88] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. **Fast exact shortest-path distance queries on large networks by pruned landmark labeling**. 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 349–360. ACM, 2013. 35

[89] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. **Fast shortest path distance estimation in large networks**. In David Wai-Lok Cheung, Il-Yeol Song, Wesley W. Chu, Xiaohua Hu, and Jimmy Lin, editors, *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, pages 867–876. ACM, 2009. 35

[90] Jon Sneyers, Tom Schrijvers, and Bart Demoen. **Dijkstra's Algorithm with Fibonacci Heaps: An Executable Description in CHR**. In Michael Fink, Hans Tompits, and Stefan Woltran, editors, *20th Workshop on Logic Programming, Vienna, Austria, February 22–24, 2006*, **1843-06-02** of *INFSYS Research Report*, pages 182–191. Technische Universität Wien, Austria, 2006. 36

[91] 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. 37

[92] Michael J. Bannister and David Eppstein. **Randomized Speedup of the Bellman-Ford Algorithm**, 2011. 37

[93] 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. 37, 56

[94] Philip N. Klein. **Multiple-source shortest paths in planar graphs**. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 146–155. SIAM, 2005. 37

[95] Sergio Cabello, Erin W. Chambers, and Jeff Erickson. **Multiple-Source Shortest Paths in Embedded Graphs**. *SIAM J. Comput.*, **42**(4):1542–1571, 2013. 38

[96] Stefan Hougardy. **The Floyd-Warshall algorithm on graphs with negative cycles**. *Inf. Process. Lett.*, **110**(8-9):279–281, 2010. 38

[97] Michael Penner and Viktor K. Prasanna. **Cache-Friendly implementations of transitive closure**. *ACM J. Exp. Algorithmics*, **11**, 2006. 38

[98] Camil Demetrescu and Giuseppe F. Italiano. **Engineering Shortest Path Algorithms**. In Celso C. Ribeiro and Simone L. Martins, editors, *Experimental and Efficient Algorithms, Third International Workshop, WEA 2004, Angra dos Reis, Brazil, May 25-28, 2004, Proceedings*, **3059** of *Lecture Notes in Computer Science*, pages 191–198. Springer, 2004. 39

[99] Sung-Chul Han, Franz Franchetti, and Markus Püschel. **Program generation for the all-pairs shortest path problem**. In Erik R. Altman, Kevin Skadron, and Benjamin G. Zorn, editors, *15th International Conference on Parallel Architectures and Compilation Techniques (PACT 2006), Seattle, Washington, USA, September 16-20, 2006*, pages 222–232. ACM, 2006. 39

[100] 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. 39

[101] 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. 40, 44

[102] Gábor Szárnyas. **Query, analysis, and benchmarking techniques for evolving property graphs of software systems**. 2019. 40

[103] Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner. *Neo4j in action*, **22**. Manning Shelter Island, 2015. 40

[104] Goetz Graefe and William J. McKenna. **The Volcano Optimizer Generator: Extensibility and Efficient Search**. In *Proceedings of the Ninth*

International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria, pages 209–218. IEEE Computer Society, 1993. 40

[105] Thomas Neumann. **Efficiently Compiling Efficient Query Plans for Modern Hardware**. *Proc. VLDB Endow.*, **4**(9):539–550, 2011. 40

[106] Thomas Neumann and Michael J. Freitag. **Umbra: A Disk-Based System with In-Memory Performance**. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. 40

[107] Alfons Kemper and Thomas Neumann. **HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots**. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206. IEEE Computer Society, 2011. 40

[108] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. **Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age**. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 743–754. ACM, 2014. 40

[109] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. **MonetDB: Two Decades of Research in Column-oriented Database Architectures**. *IEEE Data Eng. Bull.*, **35**(1):40–45, 2012. 40

[110] 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. 41

[111] 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. 41, 42

[112] The Apache Software Foundation. **Apache Spark**. https://spark.apache.org/, 2022. Accessed: 22/06/2022. 41

[113] 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. 42

[114] 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. 43

[115] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. **Worst-case Optimal Join Algorithms**. *J. ACM*, **65**(3):16:1–16:40, 2018. 43

[116] Denny Vrandecic and Markus Krötzsch. **Wikidata: a free collaborative knowledgebase**. *Commun. ACM*, **57**(10):78–85, 2014. 43

[117] Jena Team. **Jena TDB Documentation**. https://jena.apache.org/documentation/tdb/, 2022. Accessed: 23/06/2022. 44

[118] 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. 44

[119] Orri Erling. **Virtuoso, a Hybrid RDBMS/Graph Column Store**. *IEEE Data Eng. Bull.*, **35**(1):3–8, 2012. 44

[120] DuckDB Labs. **DuckDB - List**. https://duckdb.org/docs/sql/data_types/list, 2022. Accessed: 01/07/2022. 53

[121] 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. 57

[122] 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. 64

[123] Ariful Azad, Mohsen Mahmoudi Aznaveh, Scott Beamer, Mark P. Blanco, Jinhao Chen, Luke D'Alessandro, Roshan Dathathri, Timothy A. Davis, Kevin Deweese, Jesun Firoz, Henry A. Gabb, Gurbinder Gill, Bálint Hegyi, Scott P. Kolodziej, Tze Meng Low, Andrew Lumsdaine, Tugsbayasgalan Manlaibaatar, Timothy G. Mattson, Scott McMillan, Ramesh Peri, Keshav Pingali, Upasana Sridhar, Gábor Szárnyas, Yunming Zhang, and Yongzhe Zhang. **Evaluation of Graph Analytics Frameworks Using the GAP Benchmark Suite**. In *IEEE International Symposium on Workload Characterization, IISWC 2020, Beijing, China, October 27-30, 2020*, pages 216–227. IEEE, 2020. 73, 90

[124] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. **Julienne: A Framework for Parallel Graph Algorithms using Work-efficient Bucketing**. In Christian Scheideler and Mohammad Taghi Hajiaghayi, editors, *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2017, Washington DC, USA, July 24-26, 2017*, pages 293–304. ACM, 2017. 73

Appendix A

SQL vs. SQL/PGQ Queries

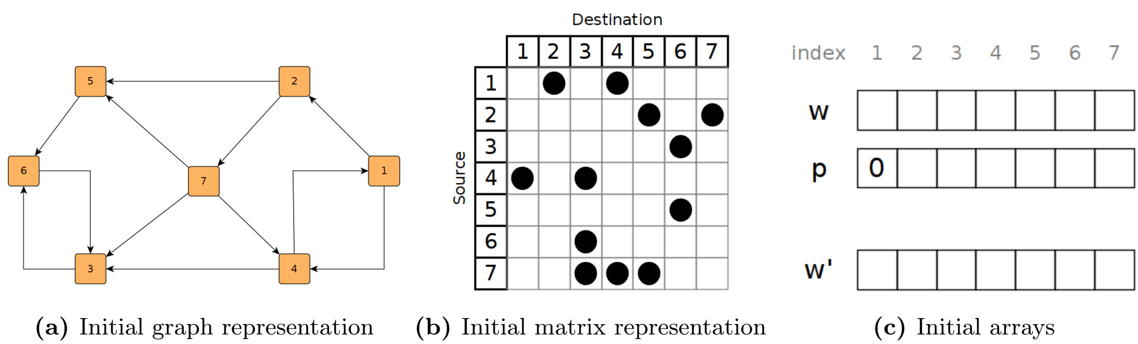
```sql
SELECT gt.c1id, gt.c2id
FROM GRAPH_TABLE ( anti_money_laundering,
  MATCH ( c1:customer )-[ t1:transfers ]->*( c2:customer )
  COLUMNS ( c1.cid AS c1id, c2.cid AS c2id )
) gt

-- The above SQL/PGQ query with a Kleene star is transformed
-- into the SQL:1999 query below
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.1: SQL/PGQ query transformed into SQL query with Kleene star

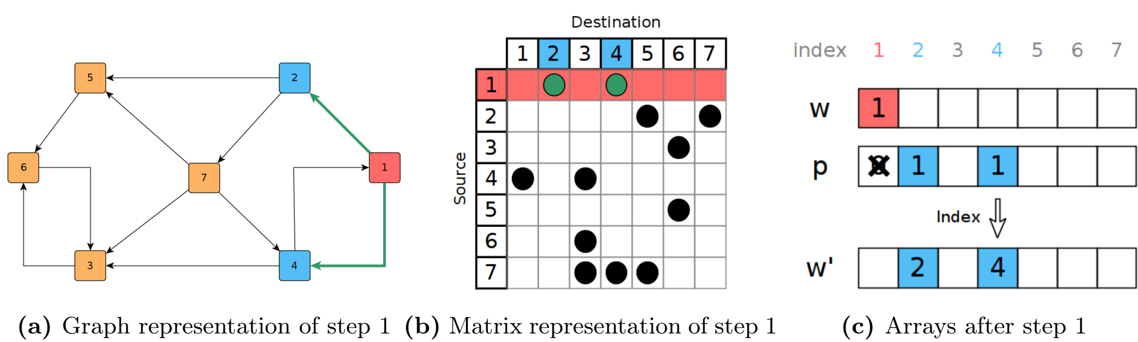
Appendix B

Any Shortest Path Example



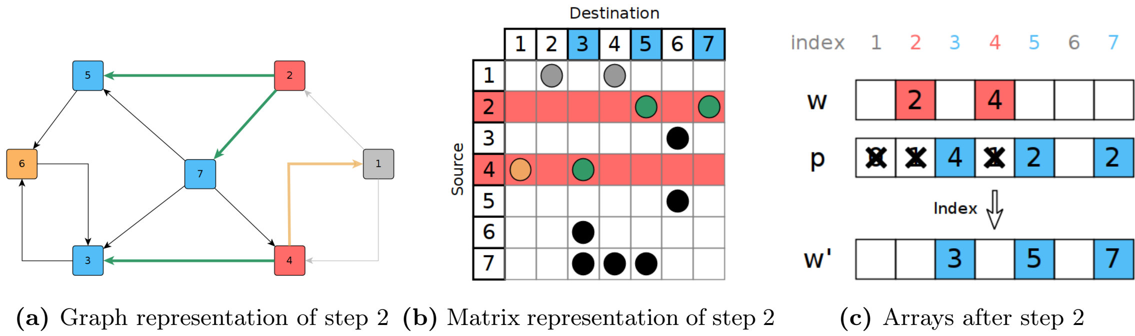
(a) Initial graph representation **(b)** Initial matrix representation **(c)** Initial arrays

Figure B.1: Any shortest path initial state



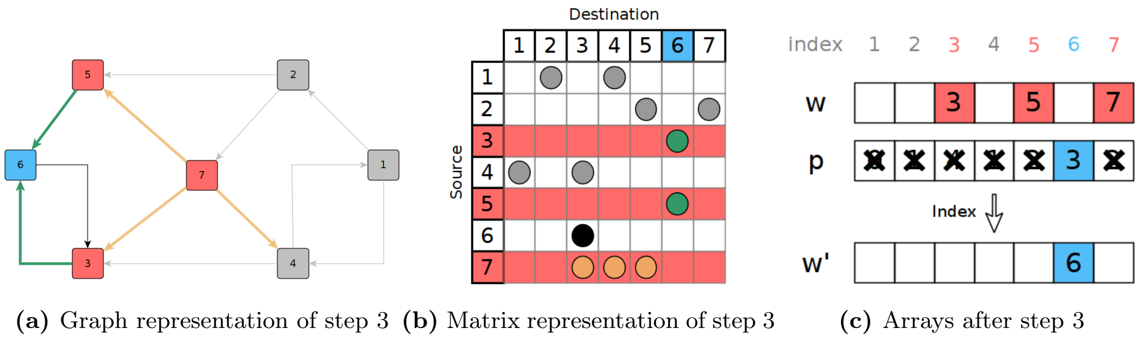
(a) Graph representation of step 1 **(b)** Matrix representation of step 1 **(c)** Arrays after step 1

Figure B.2: Any shortest path step 1



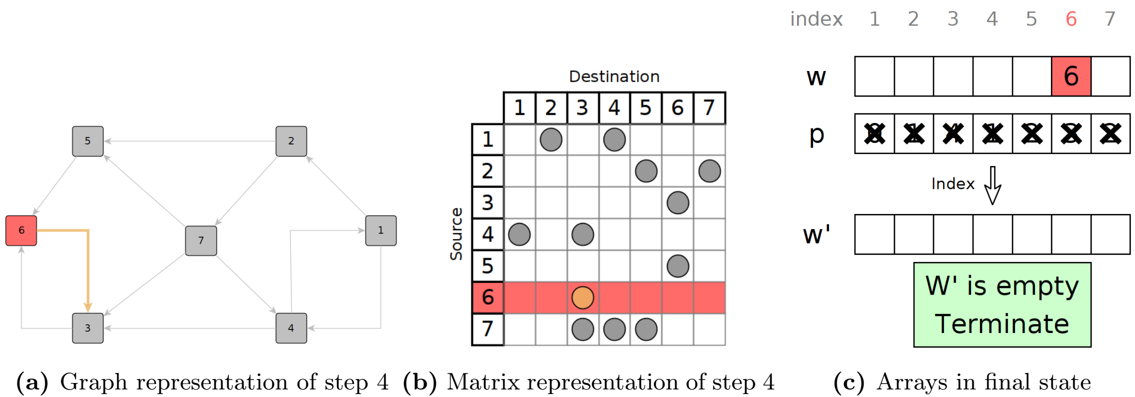
(a) Graph representation of step 2 (b) Matrix representation of step 2 (c) Arrays after step 2

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: Any shortest path 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

Summary of Literature Study

#	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	MillenniumDB: 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 [...]	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